gd

GAME DEVELOPER MAGAZINE

JUNE/JULY 1995

UBM Tech

Back

Brain Goes Whoosh!

Going to the Computer Game Developer's Conference is like having your brain ripped from your skull, spun up to 90% of the speed of light, and fired into a reaction chamber filled with 2,000 other brains moving at relativistic speeds. It's just as hard recreating the CGDC from business cards, press releases, and scrawled notes as it is seeing the signature of a Top quark in the spirals and parabolas of a reaction chamber photograph.

Above any other impression, the overwhelming support for *Game Developer* from the community was both gratifying and humbling. To have so many people known to us only from their credits on shrinkwrap games come up and tell us that they read the magazine was amazing enough, but when they continued with specific comments about this column or that feature, I realized that our magazine's editorial staff isn't what's on the masthead, it's 20,000 strong.

Despite the overload of information that makes it impossible to say what the story of the show was, I think there were three candidates: sound, 3D, and Windows.

The sound story was a howl of rage against FM synthesis and its gaming synonym "SoundBlaster compatible." To many at the CGDC, the chirpy voice of an FM card through the molded plastic speakers bundled with a $60 card represents the jackboot of tyranny on the throat of the gaming public.

Although "rendered on the fly" sounds like what happens when an accelerated brain hits a screen door, it's also where the action is in terms of software. Although demos that don't have to bother with little things like gameplay always achieve higher frame rates, vendor after vendor was showing convincing evidence that the bar's been raised (or extruded) for popular graphics.

But why fight it? The truly amazing thing is that all of this—the sound work, the three-dimensional graphics with incredible frame rates, the hottest content—was running under a single platform. Windows. You heard me right, bunkies. Microsoft has decided it wants the home market and will do what it takes to make the successor to Windows 95 (Windows 95++?) the number one game platform.

Like anyone in the press, I'm used to getting a lot of opinions about major Microsoft initiatives. You expect some people to love it on technical merit, some people to hate it on technical merit, and a lot of people to hate it on the general principle that Microsoft's a big, successful company and therefore is evil. But no matter who I talked to about the Microsoft Game SDK, with its four APIs for graphics, input, network connection, and sound, it was a love story.

Not one to be swayed by public sentiment, I put on my sunglasses with the special Anti-Hypnosis coating and met with Microsoft's Game SDK advance team. What can I say? They ripped off the glasses and made me stare at a screen with three objects composed of approximately 8,000 Gouraud-shaded polygons rotating at dozens of frames per second. Then they made me watch other displays with alpha channels, specular highlights, and smooth panning. They overlayed three-dimensional sounds. They did it all, okay? And all of this in well-behaved Windows applications.

Maybe it is hypnosis, though. I need perspective and some feedback from you, our 20,000 editors. E-mail us at gdmag@mfi.com and tell us what to do. More graphics, more Windows, more AI, more design, more channel issues? It's your magazine—tell us what to do. ■

Larry O'Brien
Editor

GAME DEVELOPER

Editor Larry O'Brien
gdmag@mfi.com
Senior Editor Nicole Freeman
76702.706@compuserve.com
Managing Editor Nicole Claro
71743.452@compuserve.com
Editorial Assistant Deborah Sommers
dsommers@mfi.com
Contributing Editors Alex Dunne
75010.2665@compuserve.com
Chris Hecker
checker@bix.com
David Sieks
dsieks@arnarb.harvard.edu
Wayne Sikes
70733.1562@compuserve.com
Editor-at-Large Alexander Antoniades
sander@mfi.com

Cover Photography Charles Ingram Photography

Publisher Veronica Costanza
Group Director Regina Starr Ridley

Advertising Sales Staff
West/Southwest
Yvonne Labat (415) 905-2353
ylabat@mfi.com
New England/Midwest
Kristin Morgan (212) 626-2498
kmorgan@mfi.com

Marketing Manager Susan McDonald
Advertising Production Coordinator Denise Temple
Director of Production Andrew A. Mickus
Vice President/Circulation Jerry M. Okabe
Group Circulation Director Gina Oh
Group Circulation Manager Kathy Henry
Circulation Manager Mike Poplardo
Newsstand Manager Pam Santoro
Reprints Stella Valdez (415) 655-4269

Chairman of the Board Graham J.S. Wilson
President/CEO Marshall W. Freeman
Executive Vice President/COO Thomas L. Kemp
Senior Vice Presidents Warren (Andy) Ambrose, H. Verne Packer, Donald A. Pazour, Wini D. Ragus
Vice President/Production Andrew A. Mickus
Vice President/Circulation Jerry Okabe
Vice President/Software Development Division Regina Starr Ridley

Miller Freeman
A United News & Media publication

Back

Delphi Why?

by Our Readers

This column is for your feedback. Send us queries, suggestions, complaints, and praise (especially praise). You want code? We've got code. (Really. Check the end of this section for a special clip-and-save piece.)

Dear Editor:

Thanks for a magazine for developers of games. I just finished reading the April/May issue and have to question your editorial. What does Delphi have to do with game development? The magazine is thin enough without extra noise being put in.

I also hope the magazine stays away from articles that compare games unless it's going to help me or someone else develop a game. Comparing Tetris on the SNES to Tetris on Sega doesn't do me any good, and I can read that in the hundreds of magazines devoted to games.

Other than these few complaints, I really like the magazine. I hope it grows and becomes very successful. I feel that this magazine will be a valuable resource for my library.

Brien King
via e-mail

Editor Larry O'Brien responds:
Close your eyes and visualize this: "Games. Windows." What do you see? A virtually empty universe, with Solitaire the brightest star. Now close your eyes and visualize this: "Windows. Home market." What do you see? The fastest growing sector in the computer industry. Vast clouds of money being sucked faster and faster into the pockets of those who create digital entertainment. Delphi is a tool for those who want to become little neutron stars of dense money.

DON'T FORGET THE MACINTOSH

Dear Editor:

As charter subscriber, I've been reading *Game Developer* since your premiere issue, and while you generally do a fine job on the article topics and writing in general, I am concerned about your blatant pro-DOS and Windows bias. (Why not just change the name to *Windows Game Developer* and avoid confusion?)

I've developed on both PCs and Macintoshes and *greatly* prefer the latter. The fact that there is a much larger market for DOS and Windows software than for Macintosh software is a tribute to Microsoft's marketing abilities. However, Macintosh developers do exist, and we would like to read some articles pertinent to us...Mode 13 and Sound Blaster programming techniques are all well and good, but how about an article on programming .mods using Sound Manager? How about "Getting the Most Out of CopyMask?"

Also, it would be a welcome addition if you'd state the platform on which a review or article is based. And please, try to get Alexander Antoniades to ease up—his comments in the April/May issue("It's a Sim, Sim, Sim, Sim World," By Design) suggested that Maxis was losing scads of money by developing on Macintoshes first. Maxis's success was probably due to the fact that it did write for the Macintosh, thereby avoiding the glut of games already available for the PC and distinguishing itself in a smaller market. Many other superior games (most notably, Myst) were developed on the Macintosh, and this platform should be given some consideration by your otherwise excellent publication.

Geoffrey Reiss
via e-mail

Larry O'Brien responds:
How about the best of both worlds, Macintosh and Windows? Jon Blossom shows how, starting on page 28.

IF I HAD MY OWN MAGAZINE...

Dear Editor:

Your magazine needs more mass to be worth buying. There are many things I'd like to see in future issues:

- Two or three long interviews per issue (8-10 pages each). There are hundreds of people with much to say—designers, businesspeople, and the like. I'd like to read about Carl Stone's, Yoshihide Otomo's, Kim Cascone's, Mason Jones's, and Fumio Kosakai's electronic music recording methods.
- I'd like a big report—interviews on Ultima 9 Development, the Japanese game scene, Psygnosis's staff, and New World's staff.
- I'd like to know what happened to Bill Hogue, Reflections, and Wizardry.
- I'd like six to ten reviews and dissections of games new and old, game structure, architecture, and the like. I've noticed that anything over two years old is being thrown off the shelves or deleted. Maybe you could print code architecture from these discontinued games.
- I'd like 30 to 40 pages of articles on porting PC code to game systems, how it's done, and the troubles and costs of hardware. I'd like four to five pages per issue of screen shots from upcoming games, works in progress, and info on CD-ROM pressing and costs.

Add in this junk and your mag will improve greatly.

D. Godat
Fort Wayne, In.

Editor Larry O'Brien responds:
Okay...we'll get right on that junk.

WELL, SOMEONE LIKES US

Dear Editor:

Thanks for such a great magazine. It is great to have articles from people who take the "theory" and put it into real world practice. In the April/May issue, I really liked Matt Pritchard's article, "Supercharging your Sprites" and Chris Hecker's column on perspective texture mapping. I also appreciate the business articles that talk about the game industry itself.

Bruce Burkhalter
Berkeley, Calif.

Editor Larry O'Brien responds:
Chris Hecker has a lot more to say on the subject. See page 18 of this issue and look for more in the future.

SOMEONE ELSE LIKES US, TOO!

Dear Editor:

I just wanted to say I like the direction in which *Game Developer* is going. In the April/May issue, you had not one, not two, but three great, informative articles—Chris Hecker's Under the Hood column, the first of two parts on texture mapping ("Perspective Texture Mapping Part I: Foundations"); "Programming Digitized Sound on the Sound Blaster" by Keith Weiner and Erik Lorenzen; and "Supercharge Your Sprites" by Matt Pritchard. Congratulations.

The only thing I didn't like were Dean Oisboid's opinions of Andre LaMothe's books *Tricks of the Game Programming Gurus* (SAMS Publishing, 1994) and *Teach Yourself Game Programming in 21 Days* (SAMS Publishing, 1994). In contradiction to what the cover says, I think these books "do suck." But that is just a difference of opinion, not the fault of your magazine.

Robert Zawarski
via e-mail

WE GOT YOUR CODE RIGHT HERE

Dear Editor:

I normally do not offer unsolicited opinions, but in this case, I will. I used to be a game developer (I programmed The Last Files of Sherlock Holmes for 3DO).

You should have it prominently displayed somewhere in your magazine that source is available via ftp. I learned this from your letters section. This could be my fault; I might have missed it somewhere. But it's a very handy piece of information.

This last thing is a compliment. Your articles were excellent—they all covered subjects that were interesting, up to date, and relevant to game programming today. There's no "Faster drawing in Mode X" stuff that was covered in *Dr. Dobb's Journal* three years ago. Texture mapping, ripping apart Tie Fighter, push-button game design...Yes! But, as I said before, give us more.

Jeff Miller
via e-mail

(clip-n-save)

❀❀❀❀

IS THIS PROMINENT ENOUGH???

❀❀❀❀

ftp://ftp.mfi.com/gdmag/src

❀❀❀❀

(clip-n-save)

Back

3D Graphics Goliaths Square Off

Alex Dunne

Microsoft's Softimage is suddenly challenged by Silicon Graphics's merger with Alias and Wavefront. What can game developers expect from these two?

Yesterday, as I was cleaning out a bookshelf in our office, I came upon an issue of *Byte* magazine from Aug., 1987. Although I was throwing everything away, I had an urge to flip through its pages—there's something compelling about a computer magazine that's over seven years old. Volume 12, number 9 of *Byte* may only have been 49 in dog-years, but it was much older in computer-years. I couldn't believe it—ads for 386 16Mhz computers selling for $4,400, 9600-baud modems for $1,000, and articles about EGA graphics. It's amazing we got through those rough times. (Some know-it-all will read this in 2002 and say the same thing about 1995, no doubt.)

One article that caught my eye focused on the technique of transferring cartoon-quality film (a clip from Disney's *Snow White and the Seven Dwarves*) into digitized EGA display. Yeeeesshhh, the final result looked horrible. So, maybe the time wasn't right back then for creating digital media from live footage. But, like a rolling snowball picking up size and speed, the graphics industry is maturing to the point where there's not too much anyone can't do at an affordable price. Microsoft and Silicon Graphics (SGI), thanks to recent acquisitions and mergers, are helping to fuel this momentum.

Competitive Partners

The relationship between Microsoft and Silicon Graphics has changed enormously over the past 12 months. Silicon Graphics is the dominant player in the graphics workstation market, and Microsoft is the giant in the PC software market. However, when Microsoft acquired Softimage last summer, Microsoft gained a powerful

Man in the boat overboard! Softimage Toonz, which was used to create this animation cel, is one product in a suite that Microsoft acquired last year. The software currently runs on the SGI platform, but Microsoft has stated its plans to port Softimage tools to Windows NT.

suite of IRIX-based animation, editing, compositing, and cel animation tools. It instantly became a key partner of SGI. Eight months later—last February—SGI merged with Alias and Wavefront, two companies that compete against Softimage on the SGI platform. How have these developments changed the relationship between Silicon Graphics and Microsoft? More importantly, how does it affect their customers?

I spoke with Andrew Wright, group product manager of advanced authoring tools for Microsoft/Softimage, and Dave Larson, director of marketing for Silicon Studios, a wholly owned subsidiary of Silicon Graphics, about the actions their companies have taken recently in the digital entertainment industry.

The most recent event, Silicon Graphics' merger with Alias and Wavefront, achieved two objectives for SGI, according to Larson.

"We felt that by merging with Alias and Wavefront," Larson explained, "we could get two of the most important groups of engineers together with our engineers and accomplish two things. [The first objective] is to drive the development of our 3D software environment... [Second,] we don't have expertise in entertainment and industrial [software] markets at the customer level like we do with hardware. We're getting a sales force that knows the customers really well at the application level, a sales force that has a much greater depth of knowledge."

What was Wright's reaction to the SGI merger?

"Surprise," he said. "From [Microsoft's] perspective, it actually puts us in a stronger position because we feel that for our customers a cross-platform solution is important. Where they want the performance of SGI, we provide it, where they want the price-to-performance ratio and openness of a Windows NT system we'll provide that to them. We'll be the only high-end 3D animation vendor that's effectively able to execute a cross-platform strategy."

I sensed no edginess from either Wright or Larson about the relationship between Microsoft and SGI, and both played up the positive aspects of their new product lines. Wright stressed the fact that many of SGI's partners, not just Microsoft, were now competitors, but that it wouldn't make sense for SGI to consider them as such: "Yes, we are a competitor to [Silicon Graphics], but they're also a competitor to a number of their other ISVs [independent software vendors]. Companies like Side Effects, Discreet Logic, Avid... One thing I can say absolutely outright is that if SGI loses their third-party applications as a result of this merger, they're dead in the water. I think they've almost got to overcompensate to make sure that their third party ISVs are treated fairly," Wright commented.

Dave Larson adamantly agreed. "We're going to treat [Microsoft] as we do a whole category of partners who will get early access information, and it's based on business parameters. These guys, as well as other 3D vendors, are still selling SGI software and we're going to do whatever we can to make sure they continue to do so. That's our business."

Softimage off the SGI Platform?

Upon acquiring Softimage last year, Microsoft stated its intention to port the Softimage tools over to Windows NT. I asked Wright whether Microsoft had plans to pull Softimage products off the SGI platform at a later date and focus exclusively on its own operating system implementation.

"No. One of the key reasons Microsoft bought Softimage is that Softimage had a tremendous presence in the community that was producing the world's best content. ILM [Industrial Light and Magic]. Greenberg. Rocket Science. For those companies, the SGI platform is absolutely critical because they need that level of performance... We think Windows NT and the associated hardware developments are going to provide a very price-attractive alternative. But in no way is that going to put SGI out of business. They are going to continue to do very well and we need to be there."

Microsoft looks at its partner/competitor relationship with SGI in the same light as its association with Apple. "We'll continue to invest in SGI," Wright stated. "It's very similar to our situation on the Macintosh. Microsoft makes a lot of money on the Macintosh and it's a very vital platform for us at the application level, even though we don't own the operating system. The fact that we've got applications on Windows 95 as well does not in any way affect our investment in the Macintosh platform."

Wright sees Silicon Graphics remaining the superior platform for high-end digital video and three-dimensional animation over Windows NT, just as the Macintosh held its position as the superior platform for graphic design when Windows 3.0 was introduced.

"Macintosh had a very strong position in graphic design. Windows came in and everybody thought that it was going to completely take over the market. As a result, companies like Aldus and Adobe developed their applications first on Windows and second on Macintosh. But they realized over time that the Mac wasn't going to go away... We think a similar thing is going to happen in the SGI world," Wright said.

Porting Softimage Products to Windows NT

Upon acquiring Softimage, Microsoft announced that it would port the company's toolset to Windows NT. Wright indicated that Softimage products would be available on Windows NT this year, but he declined to be more specific, fearing that divulging an estimated date could raise false hopes.

I wanted to know what strengths Windows NT could offer over the SGI platform to game developers. After all, SGI has been targeting this market for years and has optimized its hardware for high-end graphics and animation. Wright responded: "We think that the Windows NT platform will offer very attractive price-to-performance ratio in the range of performance that it delivers. We also feel that for people who have PC-based networks, for example developers who are using [Autodesk's] 3D Studio, it will be important for them to run a high-quality 3D product in the same environment that they're running their other tools. I think that's going to be key to the games development area."

Downward Pressure on Prices

In addition to announcing the porting of Softimage tools over to Windows NT, Microsoft announced in January that it was slashing the price of all Softimage software by up to 50%. What was behind this aggressive move? Wright explained:

"Over the last couple of years, interactive developers [have begun to] require [high-end] tools as games have become more sophisticated. We looked at our pricing structure and said, 'Well, those prices make sense if we continue to maintain our high-end feature set for our traditional market.' But if [Microsoft] really wants to penetrate the market for game developers as well as other emerging interactive media, it's important to have more aggressive price points and maintain that leadership position."

A large number of graphics and animation products have been launched for the Windows, DOS, and Macintosh platforms recently by companies like Caligari and Strata. Although these products aren't in the same class of function or performance as either the Microsoft or SGI tools on IRIX, they seem to be exerting pressure on software prices for the entire market, regardless of platform. I asked Dave Larson how Silicon Graphics viewed these lower-priced products, and how his company would respond.

"We're moving down in terms of markets," declared Larson. "As our price points come down, we're cutting deeper into various markets... Historically, SGI has been perceived as vastly more expensive and out of reach, a boutique kind of machine. We think we're rapidly expanding beyond that, and that we're within reach for a lot of people [developing digital entertainment] for a living. It's all about how much time you have to get your work done. For instance, a friend of mine just came up who's been doing a lot of audio work on the Mac, and he just started using a new audio application on our platform. He says it's dramatically affected his work just after a few days of working with it. What he used to think ahead to do he now does in real time. He can test his decisions as he goes. That's the metaphor for performance change. Everything happens so much more quickly [on the SGI platform], and your creativity can increase."

Sega and Nintendo Choose Sides

There's an interesting sidebar concerning SGI and Microsoft. The two archrivals in the game cartridge market, Nintendo and Sega, have gone to separate corners for their respective development tools, and you can probably guess whom each has enlisted. In 1994, Nintendo selected Alias (whose software was used to create the Super NES blockbuster Donkey Kong Country) as the authorized graphics development system for both current games and next-generation 64-bit games. Last January, Sega chose Softimage 3D as the official three-dimensional development tool for the new SegaSaturn game platform. I'm not saying that this is an instance of "any enemy of my enemy is my friend," but it is predictable political maneuvering.

As long as the Softimage tools on IRIX don't take a distant second priority to their Windows NT version, users stand to gain from a price war between two resource-rich companies like Silicon Graphics and Microsoft. Feature sets and performance should evolve more rapidly, and it undoubtedly will spur other SGI platform competitors to keep up.

You'd better get used to seeing more companies merging or acquired as the digital entertainment market expands—it's a natural consolidation that should continue for the next couple of years. ■

Alex Dunne is contributing editor for Game Developer *magazine.*

Back

Live from the CGDC Show Floor!

Game Developer Staff

It was some show! Alexander Antoniades kept an eye on the tenor of the show. Larry O'Brien and Barbara Hanscome checked out new products. And Nicole Claro ran into her costar from the high school play.

For those souls lucky enough to get in, the CGDC usually has enough information to make it the best show for both the seasoned and prospective game developer, and 1995 was no exception. Mushrooming up to 2,500 attendees, the CGDC irked prospective conference goers by severely underestimating demand for the second year in a row, so anyone who didn't buy tickets (or register as press) well in advance of the show has to read about it here and hope they didn't miss out on too much.

The main buzz was about the CGDC show itself and revolved around founding member Chris Crawford's talk on the second day. Crawford railed against show management and admitted that he had been kicked off the board of directors. It was a sad note to this show, which started nine years ago in someone's living room and has grown larger than anyone would have thought possible.

While last year's "unofficial" theme was the invasion of Hollywood money and fear of ratings, the definite themes this year would have to be three-dimensional development and the move to Windows. Most of the exhibitors were hawking three-dimensional tools (both sight and sound), offering to turn you into the next industry superstar if you'd just buy their tool (and the box—usually a RISC one—it runs on). On the Windows side, Microsoft explained its new write-to-the-device-drivers interface and showcased its recent acquisitions RenderMorphics and Softimage.

Lurking in the shadows, other platform vendors vied for a piece of the home entertainment pie, hoping that Microsoft might stumble in the rocky transition from DOS to Win32. Apple showed Doom and Dark Forces running native on Power Macintoshes, and IBM announced an OS/2 Warp game developer's kit and help from Argonaut and Macromedia. Dave Taylor of Id put the whole platform thing in perspective when he said the fastest version of Doom is the Linux version, so maybe there are untapped opportunities after all.

In an industry with its share of egos and superegos, the Freudian slip still crossing most developers lips remains Id. The folks from Mesquite, who received a standing ovation from awestruck developers last year, were out in force this year offering private showings of their next project, Quake, to developers deemed worthy. For an industry that uses Doom as benchmark, Quake becoming a hit is probably the only prediction that anyone can make for sure.

Low-Cost Wavetable Sound

AMD, the fourth-largest supplier of ICs in the U.S., was showing its new AM78C201 InterWave audio processor, a kick-butt, single-chip audio system. At its hospitality suite, AMD demonstrated prototype sound boards with component costs as low as $40 (implying a sub-$100 street price) running under popular games such as Tie Fighter and recreating beautiful symphonic arrangements. AMD is enthusiastic about the Microsoft Game SDK, as was Origin's Zachary Simpson, who said "The InterWave chip and the Microsoft DirectSound API will allow...unprecedented levels of audio quality and realism at consumer price points." Hoo doggie, this is one nice chip with its 16-bit, 32-voice wavetable synthesis and on-chip support for vibrato, tremolo, chorus, echo, phase shifting, and reverb. Look for boards with this chip to be available and flying off the shelves for Christmas.

For more information contact:
AMD Inc.
1 AMD Pl.
P.O. Box 3453
Sunnyvale, Calif. 94088-3453
Tel: (408) 732-2400

The O.J. Files

BioVision, of San Francisco, is a turnkey motion capture service provider. Inverse kinematics isn't time-effective for complex human motion, but with BioVision's low-cost service, you can have rotoscoped actors that can be disturbingly realistic (you might have seen BioVision's work in the frightening recreation of the murder of Nicole Simpson and Ronald Goldman that's making the rounds on the Internet) or thrilling (as in a certain wildly popular football game). For every day of shooting, BioVision estimates two days of post-processing to massage the data into formats for popular animation packages such as Alias, Softimage, Wavefront, and Nichimen Graphics. BioVision will even supply the actors! If you don't need custom work, BioVision has partnered with Viewpoint DataLabs to provide over 200 standard motion sets.

For more information contact:
BioVision
1580 California St.
San Francisco, Calif. 94109
Tel: (800) 866-3463

VIOLENCE IS FINE—BUT WHAT ELSE IS THERE?

If you were at the right place at the right time at the CGDC this year (the lobby of the Westin Hotel at 5:30 on Sunday), you might have been asked to join an informal roundtable discussing game strategies that appeal to a broader audience—namely women and nontraditional game players. Just what kind of games appeal to game players who aren't 18-to-35-year-old men? And what can game designers do to reach them?

Jessica Miller, game designer and president of Spirit Games in Salem, N.H., and myself were lucky enough to be so well positioned in the lobby, and for the next two hours we chatted with three game designers interested in bursting out of the "shoot-and-kill," model: Alex Uttermann, game designer and author of several game strategy books (including *Dragon Lore: the Official Strategy Guide*), Rusel DeMaria, game industry journalist and president of DeMaria Studio, and Dennis Hescox, vice president of Lightside Inc.

Hescox organized the roundtable in hopes to build a network, share information, and discuss marketing development and research strategies for alternative games. He came armed with psychological and academic theories and studies relating to differences between male and female game playing styles, neurophysiology, and perceptual physiology. "By understanding these differences, " Hescox explained, "we can attempt to design entertainment based on some solid ideas rather than vague and cosmetic guesses"—like pink interfaces and ponies on a box.

The conversation quickly jumped from theories to games that are simply fun to play. Uttermann, who is working on a game with DeMaria that she hopes will have an internal logic that appeals to both genders, summed it up nicely in saying, "Violence is fine in games, but what else is there?"

Puzzles that are fun to solve on their own but connect to help solve a larger puzzle; interactive games that require good communication skills and diplomacy to "win," rich plots that unfold like a novel, role-playing games, "safe environments" that allow the player to conquer the task at hand comfortably before moving on to the next challenge, and networked play were all mentioned as games and elements that would appeal to female as well as male game players.

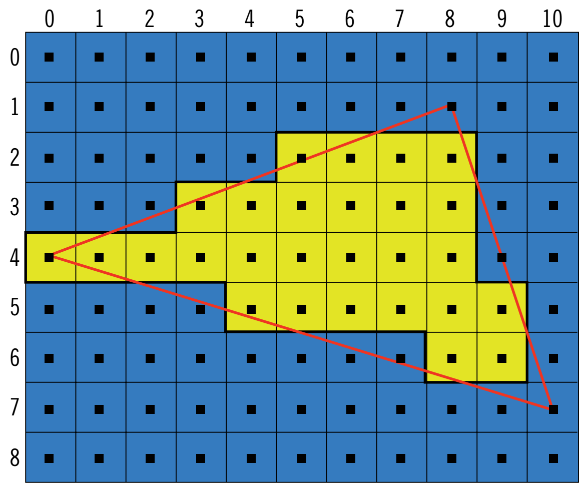
By the time we got around to how to convince the big companies in the industry to take some risks with these types of games, the hospitality suites were calling our names. But I think we could have talked all night. And I also think more people in the lobby would have joined us if they could. Perhaps we needed to hold up a big sign—if only we had a name for just what it was we were discussing. It was obvious by the end of our chat that "games for women" wasn't exactly it.

If you have thoughts and comments pertaining to women's games and alternative gaming strategies, contact Barbara Hanscome at 73611.633@compuserve.com.

Back

Perspective Texture Mapping, Part II: Rasterization

Figure 1. The Fill Convention



Did I say I'd be doing two columns? Silly me—I meant four or five columns. Our topic, perspective texture mapping, is so huge I don't know what I was thinking when I said we could cover it completely in two columns. Luckily, the topic has enough variety that it should keep everyone glued to these pages for the duration.

In Part 1, we covered most of the math behind the perspective projection and triangle gradients (those neat numbers that let us interpolate without recalculating at each scanline), and we quickly went over polygon fill conventions and stepping on pixel centers. That's a lot of information for a single article. In fact, there's so much material still to cover I'm not even going to summarize my last article beyond saying, "Read it." If you haven't read Part I, you'll still get a lot out of Part II, but you might have trouble seeing how this information fits in perspective (cough).

This time around we're going to focus on the triangle rasterization stage, and we'll expand on the math for the fill convention we derived last issue.

As I did last time, I encourage you to get out a piece of graph paper and join in the fun. Speaking for myself, I find it impossible to learn math without scribbling all over the place.

If you don't like math, well, computer graphics is math for the most part, so I'm not sure what to tell you. My goal is to describe the math in an accessible way, but I'm not going to hide the fact that math underlies everything about computer graphics, especially three dimensional computer graphics. If you like programming you will definitely like math...heck, math's even better than computer programming because there are no compiler or operating system bugs! (Of course, there's no compiler or operating system to tell you when you've done something wrong, either.)

Raster Blaster

When I say rasterization, I mean taking the continuous geometric triangle—defined by its vertices—and displaying it on the monitor's discrete display grid, or "raster." The rule we defined for doing this is called a top-left fill convention, where we light all pixels that are strictly inside the polygon boundaries and any pixels that are exactly on the polygon boundary if they're on the top or left edges (remember, pixels are boxes with a center, not just points). Figure 1 shows this fill convention in

Chris Hecker

Perspective texture mapping is a huge subject—much too big to cover in one or even two articles. In Part II of his series on the subject, Chris Hecker tackles rasterization, an essential concept.

action. Pixel (5,2) is lit, but pixel (9,4) is not, even though our polygon edge intersects both (within the limits of the magazine's printing accuracy, at least). This is because (5,2) is on a left edge and (9,4) is on a right edge. A fill convention lets abutting polygons share an edge without either polygon overwriting any pixels of its neighbor, or leaving any unlit holes—called dropouts—between the two.

A top-left fill convention for a left edge from x0,y0 to x1,y1 is defined mathematically by the ceiling function:

$$x_{\text{int}} = \left\lceil \left(\frac{x_1 - x_0}{y_1 - y_0} \right)(y - y_0) + x_0 \right\rceil$$

In my last column, I presented this equation without much explanation, so this time we'll go into it in more detail. First, we can derive the equation for the line in Figure 2 by setting the slope of the entire line equal to the slope of any line segment on that line (the segment from x,y to x_0,y_0 is on the line, so its slope is equal to the line's) and solving for x:

$$\frac{y_1 - y_0}{x_1 - x_2} = \frac{y - y_0}{x - x_0}$$

$$x = \left(\frac{x_1 - x_0}{y_1 - y_0} \right)(y - y_0) + x_0 \qquad (2)$$

We can use Equation 2 to give us the x value for any y value on the line. You can see that if $y = y_0$, then $x = x_0$ as you'd expect, and likewise for the other endpoint. This equation generates real (as opposed to integer) values for x, so we need to use our fill convention to tell us how the real x maps to an integer pixel. This is where the ceiling function comes in.

The ceiling function is defined as bumping a real value up to the next highest integer if the value has a fractional part, or leaving it alone if it is already an integer. For example:

$$\left\lceil \frac{4}{4} \right\rceil = 1$$

$$\left\lceil \frac{3}{4} \right\rceil = 1$$

$$\left\lceil \frac{5}{2} \right\rceil = 3$$

$$\left\lceil \frac{-4}{4} \right\rceil = -1$$

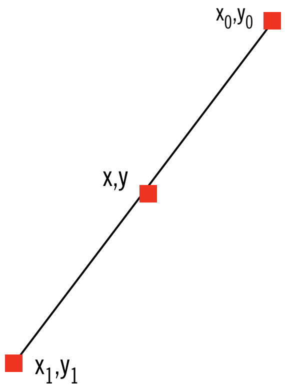
and:

$$\left\lceil \frac{-4}{3} \right\rceil = -1$$

Notice how the ceiling behaves with negative numbers—it bumps the value to the next highest value, not to the next highest absolute value.

The ceiling is the perfect function to realize a top-left fill convention for left edges (and top edges if you solve for y instead of x in Equation 2). If we're exactly on an integer pixel center we will light the pixel, but if our x is at all greater than the integer—to the left of the pixel center—the ceiling will bump us up to the next pixel that's strictly inside the edge. It should be pretty obvious that the equations for right and bottom edges are the same as for top and left edges with the addition of a minus one outside the ceiling. That is, if the edge is on the integer pixel, the ceiling won't affect it, but the minus one will knock us back one pixel following our fill convention. If the edge is greater than the pixel, the ceiling will bump it up one (to the first pixel outside the edge) and the minus one will bump it right back inside the polygon. Another way of looking at it is if we have two polygons with an abutting edge, the edge will be the left edge of one and the right edge of the other (or the top and the bottom), and they'll draw the same set of pixels, except offset by a single pixel for one polygon.

Figure 2. A 2D Line



The code to implement this was pretty straightforward in our floating point rasterizer (shown in last month's listing):

```
int XStart = ceil(pLeft->X);
```

We call the ANSI standard math.h function, `ceil()`, and use the integer returned for our starting x coordinate. As we step from one scanline to the next our real x value steps by the inverse slope, as Equation 2 shows when you set y = y + 1.

While floating-point math certainly is convenient when you're trying to get code up and running, it's probably not the best choice for a production rasterizer. First, even though floating point coprocessors are commonplace on today's machines and are even faster than the integer processor for some operations, converting from floating point to integer is still slow. Because a rasterizer is where the real three-dimensional coordinates get mapped to the integer hardware bitmap, we end up converting a lot. Also, functions like `ceil()` are actual function calls in floating point, but fall out of the math almost for free with integer coordinates.

In addition, it's hard to get the math just right for floating point numbers; there's a whole field in mathematics dedicated to figuring out how floating point numbers accumulate error. Finally, we'll see there are some benefits to using integer digital differential analyzers (DDAs) when we discuss pixel centers.

Integers from Floor to Ceiling

Before we convert our rasterizer to use integers, let's learn a couple of neat tricks for manipulating the ceiling function and its companion, the floor. The floor of a value is—you guessed it—the next-lowest integer if the value has a fractional part, or the value if it's already an integer. You could also think of this as truncating the fractional part for positive values. Following are some floor examples:

$$\left\lfloor \frac{4}{4} \right\rfloor = 1$$

$$\left\lfloor \frac{3}{4} \right\rfloor = 0$$

$$\left\lfloor \frac{5}{2} \right\rfloor = 2$$

$$\left\lfloor \frac{-4}{4} \right\rfloor = -1$$

and

$$\left\lfloor \frac{-4}{3} \right\rfloor = -2$$

Again, notice the behavior when the value is negative.

We can convert from ceiling to floor easily if a and b are integers:

$$\left\lceil \frac{a}{b} \right\rceil = \left\lfloor \frac{a-1}{b} \right\rfloor + 1 = \left\lfloor \frac{a-1}{b} + 1 \right\rfloor = \left\lfloor \frac{a-1+b}{b} \right\rfloor \quad (3)$$

Equation 3 also shows that we can move integers in and out of the floor (or ceiling). We obviously can't move fractional values in and out, though, because they can affect the result. Run through a few examples on your own to see why Equation 3 works.

Now that we have a working knowledge of floors and ceilings, let's convert the rasterizer to use integer coordinates. Because we are defining x_0,y_0 and x_1,y_1 in Equation 1 to be integers, we can manipulate the equation to our advantage. We can bring x_0 outside the ceiling function, for starters. This means any x generated by our fill convention will be the integer x_0 plus the integer result of the ceiling function for a given y. Now, let's use Equation 3 to turn the ceiling function into a floor.

Let:

$$dx = x_1 - x_0$$

and:

$$dy = y_1 - y_0$$

so:

$$x_{\text{int}} = \left\lfloor \frac{dx(y - y_0) - 1}{dy} \right\rfloor + 1 + x_0 \quad (4)$$

If our initial y value is y_0, it's easy to see the initial value in the floor is -1/dy. The floor of this is -1, and $-1 + 1 + x_0 = x_0$, as we expect. We'll be doing *forward differences* to step our edges, so after we generate the initial value for x, we're going to want to step y by 1 to the next scanline and generate the next x from our previous x value, without recalculating it from scratch. I will assume you are already familiar with forward differences, which are covered in any decent computer graphics book, so I'm just going to point out the interesting parts of this algorithm.

Perhaps the most interesting thing about this particular equation is how the floor interacts with the forward differences, especially when dx is less than 0.

Mod Squad

To thoroughly analyze Equation 4's behavior, we need another trick for manipulating floors:

$$\left\lfloor \frac{a}{b} \right\rfloor = \frac{a}{b} - \frac{a \bmod b}{b} \qquad (5a)$$

You're probably familiar with the modulus operator, `mod`, from programming in C or other languages (in C, % is the mod operator). As long as two numbers, a and b, are positive, a mod b is the integer remainder after dividing the numerator a by the denominator b. Equation 5a says we take the real number a/b and subtract its remainder over b and to get the floored value, an integer.

The mathematically defined mod usually behaves differently, in subtle ways, than the mod in your programming language of choice, and because we're using the "math-mod" in the definition of our fill convention we need to make sure we don't let an ill defined programming language muck up the works. For example, ANSI C (and C++) defines the mod operator to be the same as the math-mod operator when both operands are positive, but when either operand is negative the result is implementation dependent—the standard only defines the relationship of a/b and a%b, not their values, in this case. Fortunately our denominator, dy, is always positive because we step down the polygon from top to bottom, so we only have to deal with the case where the numerator, dx, is negative.

We saw how the floor function behaved with negative values, so if Equation 5a is true (it is, trust me), that dictates how the math-mod behaves as well. Assuming b is positive (our dy), a little thought and some scratch paper will show you that a mod b is always positive regardless of whether a is positive or negative. This is because the floor of a negative number goes to the next lowest number, so the mod term must be positive to bring it back up to the real value of a/b. Figure 3 shows a graph of x mod 3. Here's Equation 5a rearranged to make that more clear:

$$\frac{a}{b} = \left\lfloor \frac{a}{b} \right\rfloor + \frac{a \bmod b}{b} \qquad (5b)$$

Equation 5b shows a fraction as we sometimes think of it with an integer part and a fractional part, since a mod b is always smaller than b.

Even if we want to ignore the ANSI standard and hope our platform calculates mod correctly, we're out of luck on most machines, including Intel x86 processors. The x86 signed divide instruction, idiv, truncates towards 0 when dividing negative numerators, which is exactly the opposite of the real floor function. It appears we need to develop a flooring divide and mod function that works on any standard platform, that is, any platform that computes positive mods and divides correctly.

If a ≥ 0, then we'll just do the normal divide and mod. On the other hand, if a < 0, let m = (-a) mod b:

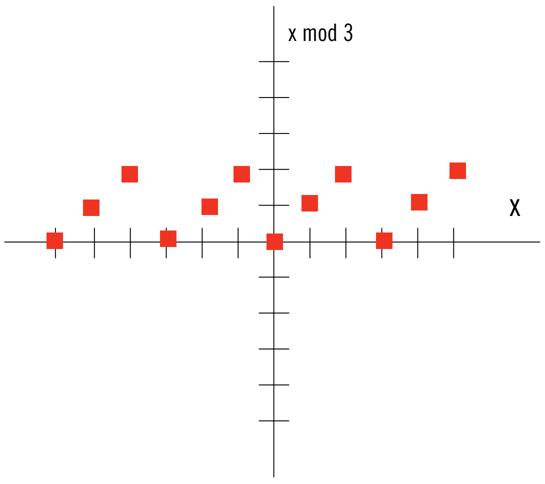
$$\left\lfloor \frac{a}{b} \right\rfloor = \begin{cases} -\left\lfloor \frac{(-a)}{b} \right\rfloor, m = 0 \\ -\left\lfloor \frac{(-a)}{b} \right\rfloor - 1, m \neq 0 \end{cases} \qquad (6)$$

$$a \bmod b = \begin{cases} 0, m = 0 \\ b - ((-a) \bmod b), m \neq 0 \end{cases} \qquad (7)$$

In other words, Equations 6 and 7 say that if m = 0 (there is no remainder), then we do the flooring divide and mod differently than if there is a remainder. This probably seems really complicated, but if you sit down with a piece of paper and refer to the equations and Figure 3 you'll see how this works in no time (okay, maybe five or ten minutes...it took me a while, too). Our C++ function to correctly compute flooring divides and mods looks like this:

```
inline void FloorDivMod( long Numerator,
 long Denominator,
long &Floor, long &Mod ) {
  assert(Denominator > 0);
// we assume it's positive
  if(Numerator >= 0) {
// positive case, C is okay
    Floor = Numerator / Denominator;
    Mod = Numerator % Denominator;
} else {
// Numerator is negative,
```

Figure 3. x mod 3



```
      do the right
    thing
       Floor = -((-Numerator) / Denomina-
tor);
      Mod = (-Numerator) % Denominator;
      if(Mod) {
// there is a remainder
        Floor--; Mod = Denominator - Mod;
      }
    }
}
```

Why?

Let's take a step back and ask ourselves (as you're probably already asking yourself), "Why do we care?" People have been rasterizing polygons since shortly after the beginning of time, and they never went through all this, you say. Well, if their polygons don't have dropouts and consistently light the correct pixels, then they went through all this or its equivalent for another fill convention.

The vast majority of rasterizers don't work properly, and that's why the vast majority of games have dropouts and overwrites at abutting polygon edges. We're taking the time up front to get the math exactly right, so we can implement our rasterizer with total confidence that it will light *exactly* the right pixels; no more, no less.

This is my personal crusade to eliminate dropouts and poor quality rasterizers everywhere, and I'm hoping you'll help me accomplish it. The best part about doing it right is it looks better and isn't any slower at run time than doing it incorrectly, there's just more to understand beforehand.

Vive La Différence

Now that we've got an algorithm for the correct divide and mod on any platform, we can go back to our original goal, which was to implement our fill convention with integer forward differences. We can use Equation 5b to manipulate Equation 4. Let n = dx(y - y_0) - 1:

$$x_{\text{int}} = \left\lfloor \left\lfloor \frac{n}{dy} \right\rfloor + \frac{n \bmod dy}{dy} \right\rfloor + 1 + x_0$$

and

$$x_{\text{int}} = \left\lfloor \frac{n}{dy} \right\rfloor + \left\lfloor \frac{n \bmod dy}{dy} \right\rfloor + 1 + x_0$$

(We can take the floor of n/dy out of the enclosing floor because it's an integer; see Equation 3.)

This is our initial state. We calculate n from our starting y value, do the flooring divide and mod (with our correct algorithm if n is negative), and use the n mod dy term's numerator as our initial error term for our forward difference. (We don't actually do the divide. It's implicit in the way the DDA functions.) Since n mod dy is positive and less than dy, we know that the floor of the n mod dy term is 0 and doesn't affect the initial x. As y steps by 1, our floor term steps by dx/dy (calculated by substituting y = y + 1 in our original equation). Our new x (call it x'), is calculated from:

$$x'_{\text{int}} = x_{\text{int}} + \left\lfloor \frac{n \bmod dy}{dy} + \frac{dx}{dy} \right\rfloor$$

We use Equation 5b on the dx/dy step to get:

$$x'_{\text{int}} = x_{\text{int}} + \left\lfloor \frac{dx}{dy} \right\rfloor + \left\lfloor \frac{n \bmod dy}{dy} + \frac{dx \bmod dy}{dy} \right\rfloor \qquad (8)$$

Equation 8 says that as y steps by 1, x steps by the floor of dx/dy, and our error term steps by dx mod dy. Note that mod is always positive, so when our error term numerator exceeds our denominator, dy, we add 1 to the resulting x regardless if we're stepping left or right. This probably differs from other DDAs you've used before—the mathematically defined floor and mod terms work out so that you're always adding 1 when your error term rolls over, not just when you're stepping in the positive direction.

Look Before You Jump

Those of you who have written fixed-point edge rasterizers instead of error-term DDAs are probably wondering why we're going to the trouble of doing a DDA, with its accompanying jumps when the error term rolls over. Even though the jump is in the scanline loop, not the pixel loop, jumps are getting more and more expensive as processors get deeper and deeper pipelines. In fact, on more recent Intel architectures the jump prediction logic makes mispredicted jumps that fall through even more expensive than jumps that are taken on earlier processors. Fear not, there is a good reason to use an error-term DDA instead of fixed-point to scan our edges.

Remember the following lines from our floating-point texture mapper

On recent Intel architectures, jump prediction logic makes mis-predicted jumps that fall through more expensive than jumps on earlier processors.

in Part 1:

```
int XStart = ceil(pLeft->X);
float XPrestep = XStart - pLeft->X;
float OneOverZ = pLeft->OneOverZ +
  XPrestep * Gradients.dOneOverZdX;
```

When we start a scanline, we need to step in to the first pixel center from the real edge before we can start drawing, and our interpolants (like 1/z in this snippet) need to step with us. We had to calculate `XPrestep` every scanline, and multiply it by the gradients of all our interpolants to get to the starting pixel center before we could draw. This is because we didn't know how far we were from the first pixel center until we did the `ceil()` call.

Now think about how this works with a DDA. We are stepping from one pixel center to the next directly, and we know exactly how far we had to come from the last pixel center: the floor of dx/dy in x plus 1 in y, or that step plus 1 in x when our error term rolls over (see Equation 8). We never need to calculate our prestep to a pixel center because we're always stepping on pixel centers! Take a minute to think this through—it means we get the advantages of sampling from pixel centers, and we don't pay the prestep multiply. As I've mentioned before, these advantages include rock solid textures that don't swim when you rotate and no "hairy texture" artifacts.

Listing 1 shows the salient parts of the integer rasterizer. Because of space constraints, I've only included the differences from last column's listing. You can pick up the entire listing on CompuServe in the *Game Developer* section of the SD Forum or from ftp://ftp.mfi.com/gdmag/src/.

The code is in a weird state because I left the texture coordinates as floats, while the edge rasterization is in integer coordinates, as we've been discussing. This bizarre combination doesn't affect the rasterizer, and it will be fixed in the next article when we address the texture mapping itself. One thing you may or may not notice when you run this rasterizer is how jerky it is compared to the original floating point rasterizer. If you

Listing 1. The Integer Rasterizer (Continued on p. 26)

```
struct edge {
    edge(gradients const &Gradients, POINT3D const *pVertices,
            int Top, int Bottom );
    inline int Step( void );

    long X, XStep, Numerator, Denominator;         // DDA info for x
    long ErrorTerm;
    int Y, Height;                                 // current y and vertical count
    float OneOverZ, OneOverZStep, OneOverZStepExtra;// 1/z and step
    float UOverZ, UOverZStep, UOverZStepExtra;     // u/z and step
    float VOverZ, VOverZStep, VOverZStepExtra;     // v/z and step
};

inline int edge::Step( void ) {
    X += XStep; Y++; Height--;
    UOverZ += UOverZStep; VOverZ += VOverZStep;
    OneOverZ += OneOverZStep;

    ErrorTerm += Numerator;
    if(ErrorTerm >= Denominator) {
        X++;
        ErrorTerm -= Denominator;
        OneOverZ += OneOverZStepExtra;
        UOverZ += UOverZStepExtra; VOverZ += VOverZStepExtra;
    }
    return Height;
}

void DrawScanLine( BITMAPINFO const *pDestInfo, BYTE *pDestBits,
    gradients const &Gradients, edge *pLeft, edge *pRight,
    BITMAPINFO const *pTextureInfo, BYTE *pTextureBits );

/******** TextureMapTriangle **********/

/********** handle floor divides and mods correctly ***********/

inline void FloorDivMod( long Numerator, long Denominator, long &Floor,
                long &Mod )
{
    assert(Denominator > 0);                       // we assume it's positive
    if(Numerator >= 0) {
        // positive case, C is okay
        Floor = Numerator / Denominator;
        Mod = Numerator % Denominator;
    } else {
        // Numerator is negative, do the right thing
        Floor = -((-Numerator) / Denominator);
        Mod = (-Numerator) % Denominator;
        if(Mod) {
            // there is a remainder
            Floor--; Mod = Denominator - Mod;
        }
    }
}

/********** edge constructor ***********/

edge::edge( gradients const &Gradients, POINT3D const *pVertices.
        int Top, int Bottom )
{
    Y = pVertices[Top].Y;
    Height = pVertices[Bottom].Y - Y;
    int Width = pVertices[Bottom].X - pVertices[Top].X;

    if(Height) {
        // this isn't necessary because we always start at TopY,
        // but if you want to start somewhere else you'd make
        // Y your start
        FloorDivMod(Width * (Y - pVertices[Top].Y) - 1,
```

compile two test programs, one with each rasterizer, and run them side by side, you'll easily see the quality difference.

The texture mapping is jerky because we use the endpoints of the triangle to compute the gradients, and the endpoints are changing by relatively large amounts as the triangle moves because of the integer truncation. You also see similar jerkiness in a lot of game rasterizers, and it's probably caused by the same thing (compounded with the artifacts generated by not stepping on pixel centers). Even in the low 320 by 200 resolution game world, this jitter is visible separately from the normal aliasing. In accordance with our quest to increase rasterization quality around the world, I find this unacceptable. The solution happens to be simple: fractional endpoints. Unfortunately, I was out of space a while back, and my editor is beginning to hate me, so the description of this solution will have to wait until next time.

Summing Up

Once again, I'm over my word budget, and I still haven't covered everything. I simply must give credit where credit is due, however—without my friend Kirk Olynyk's help and tutelage I'd still be lighting the wrong pixels without knowing the difference. If you're into this kind of discrete math (it's so useful for raster graphics) and you want to learn more, *Concrete Mathematics* (Addison Wesley, 1994) by Ronald L. Graham and Oren Patashnik is great.

Also, while discussing my article "Changing the Rules for Transparent Blts" (Under the Hood, Feb. 1995) on rec.games.programmer, Rich Gortatowsky (rg@raster.kodak.com) mentioned that for best results, your RLE compressor should try to compress vertically as well as horizontally. I totally agree.

Finally, I promise we'll get back to the actual texture mapping portion of the texture mapper next time. ■

Chris Hecker wants a single-cycle integer multiply on future x86 processors so bad he can taste it. Yum yum. You can contact him via e-mail at checker@bix.com or through Game Developer *magazine.*

Listing 1. The Integer Rasterizer (Continued from p. 24)

```
            Height,X,ErrorTerm);
        X += pVertices[Top].X + 1;

        FloorDivMod(Width,Height,XStep,Numerator);
        Denominator = Height;

        OneOverZ = Gradients.aOneOverZ[Top];
        OneOverZStep = XStep * Gradients.dOneOverZdX
                  + Gradients.dOneOverZdY;
        OneOverZStepExtra = Gradients.dOneOverZdX;

        UOverZ = Gradients.aUOverZ[Top];
        UOverZStep = XStep * Gradients.dUOverZdX
                  + Gradients.dUOverZdY;
        UOverZStepExtra = Gradients.dUOverZdX;

        VOverZ = Gradients.aVOverZ[Top];
        VOverZStep = XStep * Gradients.dVOverZdX
                  + Gradients.dVOverZdY;
        VOverZStepExtra = Gradients.dVOverZdX;
    }
}

/********** DrawScanLine ************/

void DrawScanLine( BITMAPINFO const *pDestInfo, BYTE *pDestBits,
    gradients const &Gradients, edge *pLeft, edge *pRight,
    BITMAPINFO const *pTextureInfo, BYTE *pTextureBits )
{
    // assume dest and texture are top-down
    assert((pDestInfo->bmiHeader.biHeight < 0) &&
        (pTextureInfo->bmiHeader.biHeight < 0));

    int DestWidthBytes = (pDestInfo->bmiHeader.biWidth + 3) & ~3;
    int TextureWidthBytes = (pTextureInfo->bmiHeader.biWidth + 3) & ~3;

    int XStart = pLeft->X;
    int Width = pRight->X - XStart;

    pDestBits += pLeft->Y * DestWidthBytes + XStart;

    float OneOverZ = pLeft->OneOverZ;
    float UOverZ = pLeft->UOverZ;
    float VOverZ = pLeft->VOverZ;

    while(Width-- > 0) {
        float Z = 1/OneOverZ;
        int U = UOverZ * Z;
        int V = VOverZ * Z;

        *(pDestBits++) = *(pTextureBits + U + (V * TextureWidthBytes));

        OneOverZ += Gradients.dOneOverZdX;
        UOverZ += Gradients.dUOverZdX;
        VOverZ += Gradients.dVOverZdX;
    }
}
```

Back

A C++ Class for Cross-Platform Double-Buffered Graphics

What if someone told you that you could release a game simultaneously on Windows, Macintosh, and DOS with no more effort than it takes you to release a DOS game now? Of course, you'd kick the person off your staff, find a loophole in his or her contract, or refuse the project proposal. There are no freebies in software development.

But what if someone said you could release your product on Macintosh and Windows with very little extra work and showed you how to do it?

This article takes the first step in that direction by defining a simple double-buffering architecture that will leverage your existing three-dimensional rendering, animation, sprite composition, and other custom 8-bit graphics code onto Windows and Macintosh systems. Using the small C++ class implemented here, you will be able to write 32-bit graphics applications that compile and run without changes on Macintosh System 6.0.7 and 32-bit Windows (including Win32s and Windows 95).

Double-Buffered Graphics

Double-buffered graphics provide the core graphics technology that power almost all high-performance desktop multimedia and entertainment software. From simple card games to immersive three-dimensional environments, such graphics use offscreen memory to hide image composition from the user and provide smooth transitions between frames of animation. Most games today—and probably most games created in the near future—will rely on copying or page-flipping 256-color graphics from offscreen memory to display memory.

Full-screen 8-bit double buffers such as these are chunks of RAM managed in such a way that a one-to-one mapping exists between pixels on the screen and bytes in the buffer. That's all double buffering is, so why not abstract it in a platform-independent interface?

All it takes to create an image in the offscreen buffer is a definition of the screen-to-offscreen mapping and a method to describe the image in the offscreen buffer to the screen. For all the platforms game developers generally deal with, this requires at most four pieces of information:

- `pBits`, a pointer to the offscreen buffer byte that maps to the screen point (0, 0).
- `Stride`, the number of bytes between the buffer byte that maps to the screen point (x, y) and the buffer byte representing (x, y+1).
- `Width`, the width in pixels of the rectangle represented by the offscreen buffer.
- `Height`, the height in pixels of the rectangle represented by the offscreen buffer.

Given these four pieces of information, you can implement any graphics algorithm to render three-dimensional texture-mapped environments, animate complex action sequences, or present tic-tac-toe at 30 frames per second. To make those graphics routines work on any platform, all you need is that magical interface that provides these four elemental pieces of information and a way to display completed images on the screen.

Listing 1 shows the public aspects of this dream interface in a simple C++ declaration of a class called `COffscreen-Buffer`. All operations on the class except `GetBits` are declared `const` because they do not allow change to the buffered image or to the size of the buffer. I've added the `Lock` and `Unlock` methods, which I'll explain.

Before digging into various implementations of `COffscreenBuffer`, let's take a look at how we can use such an interface. Say I have a pointer called `pBuffer` to a `COffscreenBuffer` object, and I want to set a single pixel to a given color index. From the definition of `pBits` and `Stride`, I can deduce that for any point (x, y) in the buffer such that $x \geq 0$ and $y \geq 0$ and $x <$ Width and $y <$ Height, the memory location `pBits + Stride * y + x` holds the corresponding byte of offscreen memory.

Using this definition, I can construct a `SetPixel` function using the following lines of code:

```
char unsigned *pPixel =
  pBuffer- >GetBits()
  + pBuffer->GetStride() * y + x;
*pPixel = Color;
```

Of course, this doesn't include any validation or clipping tests, and it's also not very useful in the inner loop of an optimized rendering engine. But you get the idea.

Another simple graphics function would be to fill the buffer with a solid color. Using the four atomic offscreen data, I could perform this buffer clear with a few simple instructions:

```
char unsigned *pBits =
  pBuffer->GetBits();
for (int y = 0;
  y < pBuffer->GetHeight();
  ++y, pBits += pBuffer->GetStride())
  memset(pBits, Color,
    pBuffer->GetWidth());
```

For every line in the buffer, this fills as many bytes with the specified color as there are horizontal pixels, then skips down to the beginning of the next line. If I know that the buffer occupies contiguous memory, that any additional bytes included as padding are ignored, and that `Stride` is positive, I can make a small optimization and write the buffer clear like this:

```
long BufferSize = pBuffer->
  GetHeight() * pBuffer->GetStride();
memset(pBuffer->GetBits(),
  Color, BufferSize);
```

This would, however, be a bad idea, as `Stride` has been defined as a signed `long` value and will often be negative on Windows machines.

A third simple example would be to draw a 45-degree line from the point (x, y) to the point (x+n, y+n), where n is positive. No problem:

```
char unsigned *pPixel =
  pBuffer->GetBits() + pBuffer->
  GetStride() * y + x;
// Adding Stride+1 moves the
//pointer from (x,y) to (x+1,y+1)
for (int i=0; i<n; ++i, pPixel +=
pBuffer->GetStride() + 1)
        *pPixel = Color;
```

With some extrapolation on the

Jon Blossom

Double-buffering is the heart of high-performance graphics. With this C++ class, you'll be able to write your code once and compile it for Windows or the Macintosh!

Listing 1. Public Functions

```
class COffscreenBuffer
{
public:
    // Basic information access
    char unsigned *GetBits(void);
    long GetStride(void) const;
    int GetWidth(void) const;
    int GetHeight(void) const;

    // Displaying the buffer
    void SwapBuffer(void) const;

    // Pixel access control
    void Lock(void) const;
    void Unlock(void) const;
};
```

part of the reader, these simple examples show that the COffscreenBuffer interface shown in Listing 1 provides all the essential elements for a complete graphics system. Further, none use any code outside the accepted ANSI C++, supported by any compiler worth its salt.

In other words, as long as your target platform can transfer an 8-bit packed pixel image from memory to the screen and supports an ANSI C++ compiler, you can write a COffscreenBuffer implementation and support the drawing functions above without changes.

Making It Macintosh

So let's get down to the business of implementing COffscreenBuffer on the two platforms most likely to be the multimedia systems of the future: Windows and the Macintosh. On both, I'll show you how to exploit system-supported double buffering to do everything you ever wanted—or at least start you along that path.

Color 32-bit QuickDraw introduced a new architecture for offscreen drawing support on the Macintosh called a GWorld, which became a reliable part of the operating system in version 6.0.7. This extension allows the use of QuickDraw functions to draw into structured offscreen memory, and it enables the Macintosh version of COffscreenBuffer declared in Listing 2 and implemented in Listing 3.

Constructing a buffer from a GWorld requires a single call to the handy function called NewGWorld. This API requires a rectangle describing the dimensions of the desired buffer and a color table, both of which the COffscreenBuffer constructor swipes from the active window. The dimensions come directly off the CGrafPort structure, and the color table comes from the associated PixMap. For good measure, I've chosen to lock down every handle ever used here, though you may not always have to do so.

The constructor's task finishes with the call to NewGWorld, at which point the calling application becomes the proud parent of a COffscreenBuffer object. So far, so good. However, gaining access to the bits of that GWorld proves to be a trifle difficult because the operating system has allocated the buffer in moveable memory. Enter Lock and Unlock, those traditional commands that prevent data from moving in a linear address space.

Before our application can touch the bits of the offscreen buffer, the Lock method has to guarantee that the bits won't move during pixel access. Apple provides the GetGWorldPixMap and LockPixels functions to handle this, and GetPixBaseAddr provides the magic pBits pointer when all the locking is done. Apple provides the Stride value in the rowBytes field of the PixMap structure, but the system tacks on the two high bits to make things difficult. A simple mask of 0x3FFF pulls them off the top.

In addition to locking the image in memory and masking off the high bit of the scanline offset, I've heard off and on that it's also a good idea to make sure the memory management unit is in true 32-bit access mode. SwapMMUMode handles this, storing the current mode for restoration by the Unlock method.

The Unlock method mirrors the locking function, allowing the operating system to move the offscreen buffer around when the application doesn't need it. The aptly named system call UnlockPixels handles this task, after which I reset the cached pBits value to zero to avoid being bitten by an attempt to access the bits when the buffer is

Listing 2. Macintosh Declaration

```
#include <QDOffscreen.h>  // For GWorld stuff

class COffscreenBuffer
{
public:
    // Basic information access
    char unsigned *GetBits(void) { return pBits; };
    long GetStride(void) const { return Stride; };
    int GetWidth(void) const { return Width; };
    int GetHeight(void) const { return Height; };

    // Displaying the buffer
    void SwapBuffer(void) const;

    // Pixel access control
    void Lock(void) const;
    void Unlock(void) const;

    // Constructor and Destructor
    COffscreenBuffer(void);
    ~COffscreenBuffer(void);

private:
    // Common implementation data
    char unsigned *pBits;
    long Stride;
    int Height;
    int Width;

    // Macintosh implementation data
    GWorldPtr OffscreenGWorld;
    char StoredMMUMode;
};
```

unlocked.

In debug versions, I also add an integer LockCount member variable incremented by Lock and decremented by Unlock, and I assert it is zero when the COffscreenBuffer destructor is called.

The Macintosh requires the Lock...Unlock pair, and other platforms may require it as well. All implementations of COffscreenBuffer must include both methods, whether they do anything or not, and all functions written for COffscreenBuffer must operate between a Lock...Unlock pair to be completely portable. This includes the previous examples and calls to SwapBuffer.

SwapBuffer provides the memory-to-screen transfer for the application when it finishes its graphics processing and wants to display the image. SwapBuffer will call out to CopyBits to do the job, copying the entire offscreen image into the current CGrafPort, using the visRgn as a mask. The operating system will handle all the work of clipping to the visible window area.

The COffscreenBuffer destructor is the easiest of all to implement on the Macintosh. It's just a call to DisposeGWorld, which ditches the GWorld for good.

To use the COffscreenBuffer class properly on the Macintosh, be sure to turn on the pmExplicit and pmAnimated flags of all entries in the palette of the target window to ensure that the color indices used in the offscreen buffer will properly match the colors on the screen, yielding highest copying speeds and proper color matching.

Windows

Like GWorlds on the Macintosh, WinG is the obvious candidate for implementing a double-buffering architecture for Windows. WinG lets us create a buffer, access its bits, and copy it to the screen quickly. The Windows declaration of COffscreenBuffer appears in Listing 4. The following implementation appears in Listing 5.

WinG allocates an offscreen buffer for us when we use a WinGCreateDC/WinGCreateBitmap call pair. The buffer memory allocated this way

Listing 3. Macintosh Implementation

```
COffscreenBuffer::COffscreenBuffer(void)
{
    // Use the current GDevice and GrafPort to make a GWorld
    CGrafPtr CurrentPort;
    GDHandle CurrentDevice;
    GetGWorld(&CurrentPort, &CurrentDevice);
    // Get the color table from the current port
    PixMapHandle CurrentPixMap = CurrentPort->portPixMap;
    HLock((Handle)CurrentPixMap);
    CTabHandle ColorTable = (*CurrentPixMap)->pmTable;
    // Create a new GWorld with this information
    NewGWorld(&OffscreenGWorld, 8, &CurrentPort->portRect, ColorTable,
        CurrentDevice, noNewDevice);
    // Store data that doesn´t change
    Width = CurrentPort->portRect.right - CurrentPort->portRect.left;
    Height = CurrentPort->portRect.bottom - CurrentPort->portRect.top;
    // Release the current PixMap
    HUnlock((Handle)CurrentPixMap);
}

COffscreenBuffer::~COffscreenBuffer(void)
{
    // Free the allocated GWorld
    if (OffscreenGWorld)
        DisposeGWorld(OffscreenGWorld);
}

void COffscreenBuffer::Lock(void) const
{
    PixMapHandle OffscreenPixMap = GetGWorldPixMap(OffscreenGWorld);
    if (OffscreenPixMap)
    {
        // Lock the PixMap memory and pull some info off the PixMap structure
        LockPixels(OffscreenPixMap);
        Stride = (*OffscreenPixMap)->rowBytes & 0x3FFF;
        pBits = (char unsigned *)GetPixBaseAddr(OffscreenPixMap);

        // Make sure the MMU is in true 32-bit access mode
        StoredMMUMode = true32b;
        SwapMMUMode(&StoredMMUMode);
    }
}

void COffscreenBuffer::Unlock(void) const
{
    PixMapHandle OffscreenPixMap = GetGWorldPixMap(OffscreenGWorld);
    if (OffscreenPixMap)
    {
        // Unlock the PixMap memory and reset Stride and pBits
        UnlockPixels(OffscreenPixMap);
        Stride = 0;
        pBits = 0;

        // Restore the previous MMU mode
        SwapMMUMode(&StoredMMUMode);
    }
}

void COffscreenBuffer::SwapBuffer(void) const
{
    // Copy all bits from the offscreen GWorld to the active GrafPort
    // Note: The offscreen GWorld should be locked!
    CopyBits(&((GrafPort)OffscreenGWorld)->portBits,
        &((GrafPort)thePort)->portBits,
        &OffscreenGWorld->portRect, &thePort->portRect,
        srcCopy, thePort->visRgn);
}
```

will always map to the same address. It will never move in memory as far as our applications can see, so the unneeded extra limbs Lock and Unlock can be compiled out by declaring them as empty inline functions. Only the constructor, destructor, and swap functions remain.

WinG supports both top-down and bottom-up buffer orientations, as discussed in every piece of WinG literature to date, so the first step in constructing an offscreen buffer is to determine the orientation that will make memory-to-screen blts fastest. WinG provides this information through the WinGRecommendDIBFormat function, called by the constructor before creating the offscreen buffer.

Once it has the optimal Device Independent Bitmap (DIB) orientation, the constructor fills in the biWidth and biHeight fields of the BITMAPINFOHEADER structure containing the optimal format, preserving the sign of biHeight. A color table stolen from the current system palette completes the information necessary to create a WinGBitmap and an accompanying WinGDC, which the COffscreenBuffer constructor does for us. Selecting a WinGBitmap into a new WinGDC pops out a stock monochrome bitmap that the destructor will need later, so I store it in the COffscreenBuffer object until then.

The Width and Height of the buffer came from the foreground window, and WinGRecommendDIBFormat returns the pBits for the buffer. Only the Stride remains, and it's easily calculated because we know the WinG buffer is actually a standard Windows DIB. Every scanline of a DIB begins on a 4-byte boundary, and since our offscreen buffers are one byte per pixel, Stride is just the DWORD aligned Width:

```
// Align to the highest 4-byte boundary
Stride = (Width + 3) & (~3);
```

If WinG recommends a top-down DIB, that's all the calculation needed to set up the buffer. pBits points to the top of the buffer, coinciding with (0,0), and Stride indicates a positive step through memory.

For bottom-up DIBs, however, things must be switched around. As it stands, pBits would point to the last scanline in the buffer, and Stride would be the offset from (x, y) to (x, y-1). If you've used WinG before, you'll know that flipping these values around to point in the correct direction requires only two lines:

```
// Point to first scanline (end
   of buffer)
pBits = pBits + (Height - 1) * Stride;
// Orient Stride from bottom to top
Stride = -Stride;
```

With that, the constructor has finished its work, and the calling application is the happy owner of a WinG-based offscreen buffer, wrapped in a COffscreenBuffer object. Your application can draw into this buffer however you choose, calling SwapBuffer when you're ready to display an image.

The Windows SwapBuffer implementation grabs the Device Context from the active window and uses a straightforward WinGBitBlt call to copy the image from the offscreen WinGDC to the topmost window on the screen. Nothing could be easier.

The call to GetDC returns a virgin device context that will not reflect selections you may have made to previous DCs from the window. It contains no palette information. If you have gone through the effort to create an identity palette for the window (or are using the WinG halftone palette), your work will be in vain unless you register the Window with the CS_OWNDC style, which preserves device context settings over GetDC...ReleaseDC call pairs.

When it comes time to destroy the COffscreenBuffer, only three steps need to be taken: selection of the original mono-

Listing 4. Windows Declaration

```
#include <wing.h>  // For WinG stuff

// Note that this is supposed to be for Win32, so there are no FAR types.
// However, it could be adapted easily for 16-bit Windows.

class COffscreenBuffer
{
public:
    // Basic information access
    char unsigned *GetBits(void) { return pBits; };
    long GetStride(void) const { return Stride; };
    int GetWidth(void) const { return Width; };
    int GetHeight(void) const { return Height; };

    // Displaying the buffer
    void SwapBuffer(void) const;

    // Pixel access control - these are no-ops in Windows
    void Lock(void) const {};
    void Unlock(void) const {};

    // Constructor and Destructor
    COffscreenBuffer(void);
    ~COffscreenBuffer(void);

private:
    // Common implementation data
    char unsigned *pBits;
    long Stride;
    int Height;
    int Width;

    // Windows implementation data
    HDC OffscreenDC;
    HBITMAP OffscreenBitmap;
    HBITMAP OriginalMonoBitmap;
};
```

chrome bitmap back into the `WinGDC`, destruction of the `WinGBitmap`, and destruction of the `WinGDC`.

Extending the Buffer Class

The `COffscreenBuffer` class I've developed here provides only the most basic elements of a double buffering system, barely enough to be useful. Many other features could make it a very useful tool in cross-platform game development, but I have left them out for the sake of brevity.

One important thing missing from this `COffscreenBuffer` class is the explicit connection of an offscreen buffer to a window. As implemented here, `COffscreenBuffer` uses whatever window happens to be active at the time a method is invoked. When your application isn't in the foreground, this can be a messy thing!

It's easy to store a platform-specific window identifier in the `COffscreenBuffer` structure, and you can hang a pointer back to the buffer object on the window, too. Under Win32, try using `Set/GetWindowLong` and `GWL_USER` to store the pointer. On the Macintosh, `Set/GetWRefCon` performs a nearly identical task.

Of course, attaching a buffer to a resizable window means you'll have to do something smart when the window changes size. Matching the buffer dimensions to the new dimensions of the window wouldn't be a bad idea.... You'll need to look at `UpdateGWorld` on the Macintosh, and you'll most likely have to create a new `WinGBitmap` under Windows.

Many applications don't want an API as clumsy as `SwapBuffer`. You want to optimize your screen accesses by writing only the areas that haven't changed. A `SwapRect` method would do the trick very nicely. Implement it on both platforms, and remember to make the rectangle description platform independent!

And what about colors? Both constructors use the current palette to initialize the offscreen color table, but wouldn't it be nice to enable color animation? Just be sure to maintain that 1:1 offscreen mapping for speed.

Listing 5. Windows Implementation (Continued on p. 34)

```
// This is here to keep it off the stack during the constructor call
struct {
    BITMAPINFOHEADER    Header;
    RGBQUAD             ColorTable[256];
} BufferInfo;

COffscreenBuffer::COffscreenBuffer(void)
{
    HWND ActiveWindow = GetActiveWindow();

    // Make the buffer the same size as the active window
    RECT ClientRect;
    GetClientRect(ActiveWindow, &ClientRect);
    Width = ClientRect.right - ClientRect.left;
    Height = ClientRect.bottom - ClientRect.top;
    Stride = (Width + 3) & (~3);

    // Set up the header for an optimal WinGBitmap
    if (WinGRecommendDIBFormat((LPBITMAPINFO)&BufferInfo))
    {
        // Preserve sign on biHeight for appropriate orientation
        BufferInfo.Header.biWidth = Width;
        BufferInfo.Header.biHeight *= Height;

        // Grab the color entries from the current palette
        HDC hdcScreen = GetDC(ActiveWindow);
        if (hdcScreen)
        {
            PALETTEENTRY Palette[256];
            GetSystemPaletteEntries(hdcScreen, 0, 256, Palette);

            ReleaseDC(ActiveWindow, hdcScreen);

            // Convert the palette entries into RGBQUADs for the color table
            for (int i=0; i<256; ++i)
            {
                BufferInfo.ColorTable[i].rgbRed = Palette[i].peRed;
                BufferInfo.ColorTable[i].rgbGreen = Palette[i].peGreen;
                BufferInfo.ColorTable[i].rgbBlue = Palette[i].peBlue;
                BufferInfo.ColorTable[i].rgbReserved = 0;
            }
        }

        // Create the offscreen DC
        OffscreenDC = WinGCreateDC();
        if (OffscreenDC)
        {
            // Create the offscreen bitmap
            OffscreenBitmap = WinGCreateBitmap(OffscreenDC,
                (LPBITMAPINFO)&BufferInfo, (void * *)&pBits);

            if (OffscreenBitmap)
            {
                // Adjust pBits and Stride for bottom-up DIBs
                if (BufferInfo.Header.biHeight > 0)
                {
                    pBits = pBits + (Height - 1) * Stride;
                    Stride = -Stride;
                }

                // Prepare the WinGDC/WinGBitmap
                OriginalMonoBitmap = (HBITMAP)SelectObject(OffscreenDC,
                    OffscreenBitmap);
            }
            else
            {
                // Clean up in case of error
                DeleteDC(OffscreenDC);
```

Wrapping it Up

To compile the code presented in this article, I used Microsoft Visual C++ 2.0 on Windows NT version 3.5 and Symantec C++ 7.0 on Macintosh System 7.0. I ran the applications created on Windows 3.11 with Win32s, Windows NT v. 3.5, and Macintosh System 7.0. The graphics code implemented on top of the `COffscreenBuffer` API did not change.

If multiplatform graphics programming can be this easy, the excuses for writing DOS-only games begin to look silly. Take, for example, that it took only a weekend (with no sleep) to create a graphics-only version of Doom for Windows using WinG because of the structured design chosen by the programmers at id Software. Most of the third-party three-dimensional rendering systems on the market today have at least three versions—DOS, Windows, and Macintosh. Why shouldn't you?

The Microsoft machine has finally started cranking up and facing the problems of providing real game support in Windows, much of it promised for Windows 95. More and more Macintosh games are appearing on the market, and maybe one day someone will actually come up with a decent Macintosh joystick. DOS continues to score with game programmers, but users hate the configuration problems.

Using a simple system like the `COffscreenBuffer` class introduced here will enable all of your existing graphics routines on all three platforms and may help you reach more users. With the advent of Windows 95 and its promised support for sound mixing and joystick input, and with the multimedia capabilities already provided on Mac and Windows, your reasons for sticking exclusively to DOS begin to look shortsighted. Why not take the cross-platform plunge? ■

Jon Blossom is the coauthor of the WinG graphics library for Windows and is the author of Gossamer, a free three-dimensional polygon engine for the Macintosh. He currently works for Maxis and can be reached at blossom@mobius.net or through Game Developer *magazine.*

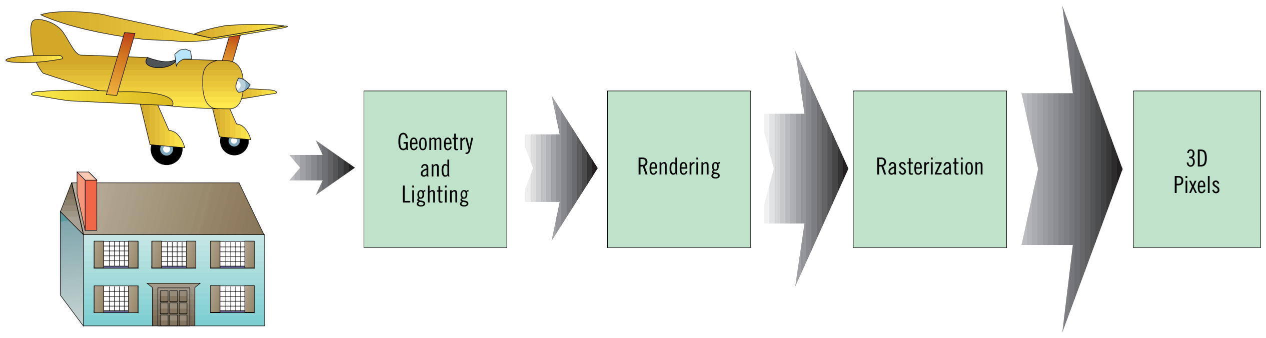
Listing 5. Windows Implementation (Continued from p. 33)

```
                OffscreenDC = 0;
            }
        }
    }
}

COffscreenBuffer::~COffscreenBuffer(void)
{
    // Delete the offscreen bitmap, selecting back in the original bitmap
    if (OffscreenDC && OffscreenBitmap)
    {
        SelectObject(OffscreenDC, OriginalMonoBitmap);
        DeleteObject(OffscreenBitmap);
    }

    // Delete the offscreen device context
    if (OffscreenDC)
        DeleteDC(OffscreenDC);
}

void COffscreenBuffer::SwapBuffer(void) const
{
    // Use the DC of the active window
    // NOTE: You'll lose the 1:1 palette mapping if the Window isn't CS_OWNDC
    HWND ActiveWindow = GetActiveWindow();
    if (ActiveWindow)
    {
        HDC ActiveDC = GetDC(ActiveWindow);
        if (ActiveDC)
        {
            // Perform the blt!
            if (ActiveDC)
            {
                WinGBitBlt(ActiveDC, 0, 0, Width, Height, OffscreenDC, 0, 0);
                ReleaseDC(ActiveWindow, ActiveDC);
            }
        }
    }
}
```

Back

Game Play Benchmarks for 3D Graphics

Figure 1. The Three-Dimensional Graphics Pipeline



Imagine you have just purchased the top-of-the-line three-dimensional game board for your PC. You find yourself playing a flight simulator with texture-mapped polygons and digital sound effects. You can hear the roar of the engines as you fly over the San Mateo bridge. You execute a steep dive, and, squeezing down on the shaking joystick, you blast away at the terminals spreading havoc amongst the innocent.

You've fed your lust for destruction. Then you hear a voice say that Moffett Field has just launched interceptor F14s to hunt you down. Let the missiles fly—you're ready for a dogfight. At this point, you don't really care that the hills are texture-mapped polygons using real Landsat imagery. You are much more interested in surviving the upcoming dogfight. Unfortunately, as you pass over the Oakland hills at 500 feet, the frame rate goes down, and your game turns into a major drag. What happened?

Frame Rate Blues

The frame rate fell below one frame per second because the game accelerator's graphics capability was unable to handle the number of polygons in the scene. How do you avoid buying a graphics card that makes a game unplayable? What measurements or criteria can we use to judge the performance of a particular game on a particular platform?

These questions are of concern to many game developers as well as players. The answer is a standard benchmark that's related to the game rather than the hardware pixel memory rates. But a standard benchmark implies a normalized level of understanding. In this article, we'll explore the technology associated with measuring game play.

Three-dimensional graphics are the next great graphics hurdle for PCs and console platforms. For the past two decades, the workstation industry has been building three-dimensional graphics engines for CAD/CAM that range

Stephen Johnson

Making three-dimensional games that are playable is no easy task. It's up to the industry to create standards and benchmarks so consumers aren't confused by false magic number models.

in cost from $20,000 up to millions of dollars. Suddenly that technology is available in a PC or a game console platform.

When a new technology hits the streets, the first things we hear are performance quotes, which are usually exaggerations, and downright lies. Open up any gaming magazine, and you can read about three-dimensional graphics performance. For example, a recent article quoted that a certain console box could do 1 million flat-shaded polygons per second and 500,000 texture-mapped polygons per second. What do these numbers really mean? What is flat shading, and why is it mentioned? How do these numbers compare to the 100,000 polygons per second that a $20,000 workstation can do? Is it actually true that a $300 game console is 10 times faster in three-dimensional graphics then a $20,000 workstation? Obviously, something must be wrong. Let's look at the problem by digging a little deeper into three-dimensional graphics processing and benchmarks.

Graphics Processing

Let's define a three-dimensional scene that contains objects from the real world. A good example would be a room with walls, a floor, table, chair, and TV. Each object is broken down into polygons. We've given the polygons position information in terms of width, height, and depth, which mathematicians and computer graphics professionals usually refer to as x, y, and z respectively.

Application software must turn each polygon in each of the objects in the scene into pixels in the frame buffer for visual consumption by an observer. This process of converting objects into polygons is called tessellation. The process of converting polygons into pixels is called the three-dimensional graphics pipeline. The pipeline is complex and difficult to understand without a good understanding of geometry and maybe a little physics. Most of the details aren't necessary for this discussion, but it is necessary to understand that these steps exist and they affect the overall performance measurements of a three-dimensional application. A simplified version is shown in Figure 1, where the arrows show the flow of data from one stage to another. The larger the arrow, the larger the amount of data flow. The greatest amount of data is passed from the rasterization step to the pixels step.

Suppose the house in our scene is represented by 1,000 polygons. When these polygons are displayed on a 1,280-by-1,024-pixel display, the actual number of pixels processed is very likely to be in the millions. This implies that the majority of the time in the graphics pipeline is spent generating pixels—and this is true. Over 80% of the time spent in the graphics pipeline is spent in the rasterization stage generating pixels. To accelerate three-dimensional graphics, first-generation chip designers are accelerating the rendering and rasterization steps. This produces the highest performance gain for the least amount of investment. The Glint chip from 3D Labs is a perfect example of a first-generation hardware solution. This chip converts triangles from x,y,z points into pixels, using a 64-bit memory interface.

Three-Dimensional Pixels

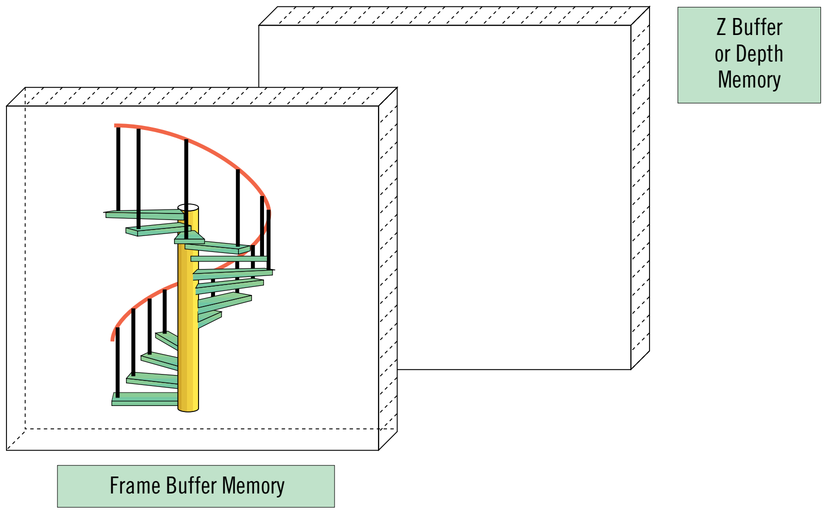
Pixels are generated for the frame buffer by flat shading, Gouraud shading, or texture mapping. Flat shading fills a polygon with pixels that are all the same color. Gouraud shading fills a polygon with pixels that are computed using color gradients. Texture mapping fills a polygon with a two-dimensional bitmap like a digital photograph.

Each of these methods has its merits. Flat shading is very fast, but it doesn't produce a very good simulation of the real world. Gouraud shading produces a much better representation of a real-world object, and it too is fairly fast. Most three-dimensional graphics hardware can produce almost exactly the same performance with flat- or Gouraud-shaded polygons. Texture mapping produces very realistic effects, but it is much slower than flat or Gouraud shading.

The 3DO console and Doom use texture mapping—in the case of the game, the technique makes players feel like they are in a real three-dimensional environment. Unfortunately, for various reasons, texture mapping is slow, and poor quality texture mapping produces a variety of nasty artifacts unacceptable to players.

So, you can generate a three-dimensional pixel using one of these techniques or any combination of them. Figure 2 shows a three-dimensional pixel, which is the depth value and the x, y location taken together. The z value is stored into a z buffer. A z buffer is a large chunk of memory dedicated to saving the last z or depth value for a particular x, y location in the display. For every pixel in the display, there is a z or depth value. For a 1,280-by-1,024 display, there exist 1,310,720 pixels in the frame buffer and the same number of depth values. The biggest problem with a z buffer is that it increases the amount of memory on the display subsystem, and this cost is passed onto the consumer by a price increase.

Figure 2. 3D Pixels Plus Frame Buffer and Z Buffer



What's in a Benchmark?

Traditionally, benchmarks comprised small applications that measure the performance of a machine. Dhrystones and Whetstones are perfect examples of this. These benchmarks contain a variety of application-like programs that run in a certain period of time on a particular platform.

The time it takes to execute these programs on a particular platform is measured and can be compared on a relatively fair basis to the time it takes to execute them on another platform. Usually, the time measured is used to compute a standard unit number. The industry as a whole looks over the programs in the benchmark and decides in some sense what is fair and what is not fair.

A similar concept exists in the current PC graphics benchmark area. Benchmarks execute a series of graphics instructions using a standard application programmer interface (API) like the graphics device interface (GDI) in Windows. The time it takes to perform the operation is measured. The measured time is used to determine how many operations the platform can do per second. So, numbers like 40 million lines per second for a benchmark that measures line rendering using GDI are reasonable.

Some benchmarks execute real applications in a scripted environment resulting in a time measurement. These benchmarks often produce a single magical number that can be used for comparison.

Benchmarks are not perfect. They don't actually measure what a particular user is doing on a particular day. Instead, benchmarks use a standard selection of operations that are supposed to represent the core activity of the average user. Thus, the goal of a benchmark is to reduce the potentially infinite combination of operations (CPU instructions or graphics primitives) down into a subset that represents a reasonable cross-section of application or user activities. Presumably, the representation of user activity is fair across the variety of platforms available.

In the workstation arena, the three-dimensional benchmark is Graphics Performance Criteria (GPC). It took 10 years and hundreds of engineers to implement this benchmark. The result is a magic number that you can easily translate into frames per second. GPC is essentially an application environment with database traversal, lighting, clipping, and rendering. On PC platforms there exists a large number of GPC-style applications—that is, games. Most of the games in the PC already output frames per second. A perfect example of this is Domark's Flight Simulator Toolkit, which lets the user display the performance levels while the game is running.

Current Benchmarks

In terms of three-dimensional graphics benchmarks, do the 1 million flat-shaded polygons per second really tell us anything about the performance of the console box? A number of key factors are missing from such a claim, such as the database traversal, geometric transformations, clipping, and lighting. This performance claim relates to the final stage of the three-dimensional pipeline, the rasterization stage. This stage only measures the performance of the frame buffer memory subsystem.

Chip designers and developers quote the memory subsystem bandwidth in pixels per second, which you can easily convert to polygons per second once you know the area of the polygon. If the area of the polygon is quoted with the polygons per second, at least we can better understand the rasterization stage in the pipeline.

This element is missing from the quote of 1 million polygons per second. The performance claims for the console box and the workstation mentioned previously should be modified to 1 million flat-shaded, single-pixel polygons per second vs. 100,000 flat-shaded, 100-pixel polygons per second. Now we can compare the last portion of the pipeline on a relatively fair basis and then make a straightforward determination that the workstation is 10 times faster than the console.

Given the area of the polygons, we can compare the performance of the memory subsystem. A fair comparison between two platforms can only be made if the other stages have been computed. Then a frames per second number can be computed.

Game Play Benchmarks

A unit of measurement for graphics rendering must address the application level processing that takes place. This is consistent with benchmarks that exist for CPUs (Dhrystones and Whetstones) and for GDI (WinBench). Frames per second is a perfectly good unit of measurement for game play.

With this type of performance metric, it is possible to say one platform runs Doom II at five frames per second, and another runs it at 60 frames per second. The higher-performance Doom II platform is obvious. With frames per second, there exists a system-level performance measurement that is reliable for comparing different platforms.

The performance of rendering one scene is computed by taking the various boxes shown in Figure 1, measuring the performance of each box, and summing the results into one aggregate number. The separate numbers are as vital as the total result because they give us the ability to understand where the bottlenecks are and optimize the graphics pipeline to achieve higher performance for the application.

Unfortunately, the hardware manufacturer provides only the polygons-per-second statistic, and that isn't enough. We only capture the last stage in the three-dimensional application pipeline. What other information can the manufacturer supply that will allow us to interpolate a frames-per-second statistic?

Playing the Game

A fair benchmark and the subsequent published results allow the consumer a level of confidence. They also allow the run-time application to choose an appropriate level of complexity in each frame at run time. For the actual user, it appears that a "magic" number is most appropriate, but this number must be based on a benchmark that exercises the entire graphics pipeline. Game developers must insist that a benchmark can be easily translated into a metric like "30 frames per second at 1,024-by-768 at 256 colors per pixel."

The magic number model can be published with the graphics board or the actual game platform. These numbers give the consumer some guarantee of game play performance. This is the same model that is applied to Windows performance today. If a graphics board or system has a WinBench 95 of 15 then the consumer knows the platform is a good accelerator for the majority of Windows applications. Also, if another graphics board is a level 19, then there is some amount of confidence that the later model is faster for most Windows applications.

The magic number also has another effect, which isn't necessarily seen by the consumer. If the operating system environment saves the performance criteria for the platform into system information, then the game can scale its displayed data appropriately. For example, if a certain platform produces 10,000 texture-mapped polygons per second through the benchmark then the application can choose to scale back the scene complexity to match the platform metrics. You can easily accomplish this by storing the results of the benchmark in a system file and making it accessible to the application. Microsoft has already announced its use of this strategy to the game developer community.

This system-wide performance information is very important, as certain games show us. For example, Tie Fighter lets the player activate an options panel to change the rendering pixel complexity at run time. Gouraud shading and other effects can be turned on or off, depending on the player's choice. The game runs in a default mode that is aimed at a 486 DX2/66. With system-level performance information stored in an easily accessible form, Tie Fighter can automatically choose the appropriate level of complexity using the benchmark results for the platform.

This doesn't remove the necessity for the options panel, but it does allow the application to start out in a more appropriate complexity level. At runtime, the game software reads the benchmark information and relates it to database complexity and rendering performance, thus determining a "best guess" at the frames-per-second number, which determines the game play criterion. This idea can also be applied to other areas of the platform. System-level performance data can cover CPU, video, CD-ROM, audio, fax/modem, and other areas to provide the application with complete performance information about the run-time environment.

Conclusion

It is very difficult today to choose a platform that runs the latest and greatest three-dimensional game with any certainty that the game will be playable. This is a problem the PC industry must solve. Game developers and large companies such as Microsoft are working on a solution. I mentioned that the benchmark solution for the PC should result in run-time game complexity adjustment based on a system information accessible to the game. In addition, the benchmark results for platforms and graphics cards should be published and understood by users.

Obviously, the best benchmark for a game is the actual game itself, but that isn't enough. Some type of standard benchmark criteria must exist that game developers and players can rely on for fair comparison between platforms. The current performance numbers of polygons per second usually only measure the memory subsystem performance. It is possible to guess at a frame-per-second number from a polygons-per-second quote if the area in pixels is also mentioned. Thus, chip vendors must be encouraged to publish numbers that are fully qualified, like "100,000 lighted, clipped, texture-mapped, 50-pixel polygons per frame at 65,536 colors per pixel on a PPC 604 processor." With this quote, a game developer can make an intelligent guess as to the resolution and complexity of each frame in relation to the game.

The PC industry should adopt a standard benchmark for games and applications that use three-dimensional graphics. The benchmark should be similar to the GPC benchmark used in the workstation industry in that it should measure the application-level performance and not just the memory subsystem of the graphics device. If a "magic" number is used then it should easily translate into scene complexity and the various stages of the graphics pipeline. This magic number information should be stored with the platform in an easily accessible form so that games can adjust scene complexity during run time. ■

Stephen Johnson has been a computer software engineer for the last 15 years. He has extensive background in C, C++, and several different assembly languages as well as experience with UNIX, MacOS, MSDOS, and Windows. About a year ago, a game called DOOM popped up on his home computer. After one intense weekend of blowing away demons, he decided to change his career. Strangely enough he discovered that writing game software is way more fun than CAD/CAM drafting packages. He currently resides at Diamond Multimedia, where he is bringing high-resolution, high-performance three-dimensional graphics to every Windows user he can find.

Getting Started with VESA Graphics

Table 1. VESA Mode Numbers

VESA MODE NUMBER	TYPE	RESOLUTION	COLORS
100h	GRAPHICS	640x400	256
101h	GRAPHICS	640x480	256
102h	GRAPHICS	800x600	16
103h	GRAPHICS	800x600	256
104h	GRAPHICS	1024x768	16
105h	GRAPHICS	1024x765	256
106h	GRAPHICS	1280x1024	16
107h	GRAPHICS	1280x1024	256
108h	TEXT	80x 60	16
109h	TEXT	132 x 25	16
10Ah	TEXT	132 x 43	16
10Bh	TEXT	132 x 50	16
10Ch	TEXT	132 x 60	16
10Dh	GRAPHICS	320x200	32K
10Eh	GRAPHICS	320x200	64K
10Fh	GRAPHICS	320x200	16M
110h	GRAPHICS	640x480	32K
111h	GRAPHICS	640x480	64K
112h	GRAPHICS	640x480	16M
113h	GRAPHICS	800x600	32K
114h	GRAPHICS	800x600	64K
115h	GRAPHICS	800x600	16M
116h	GRAPHICS	1024x768	32K
117h	GRAPHICS	1024x768	64K
118h	GRAPHICS	1024x768	16M
119h	GRAPHICS	1280x1024	32K
11Ah	GRAPHICS	1280x1024	64K
11Bh	GRAPHICS	1280x1024	16M

* 32K color modes have 1 unused bit, and 5 bits of Reg,Green, and Blue data.

* 64K color modes have 5 bits of Red and Blue data and 6 Bits ofGreen data.

* 16M color modes (true color) have 8 bits each of Red, Green, and Blue data.

Have you seen some of the games that have come out recently? A new trend is afoot in the game industry. Games are moving away from 320-by-200-pixel graphics to 640-by-480 and higher resolutions. For a long time now, we've seen game after game with a screen resolution of 320-by-200 pixels—first with EGA's 16 colors and then VGA's 256 colors. In this article I'll introduce you to the force that's making this upward graphics mobility in games possible and practical: VESA BIOS extensions.

In the beginning of the PC world, IBM would introduce a display adapter, and it would automatically become the standard to which developers wrote programs. First came the CGA in 1982, followed by the EGA in 1985, and the MCGA and VGA in 1987. Games and other graphics programs adapted to the new video standard, compilers added built-in support, and the world was an orderly place for developers.

So what happened to produce the video card Tower of Babel that we have today? IBM brought out the 8514/A display adaptor and users and developers gave it a big thumbs down.

But the maturing PC industry didn't sit still for IBM. In the absence of a standard that the market was eager for, video card manufacturers rushed in to provide their own solutions. First came enhanced EGA cards with resolutions up to 800-by-600 pixels, which were quickly swept away by a new wave of VGA-compatible cards that took full advantage of analog VGA monitors, which were rapidly replacing TTL digital monitors. Manufacturers rushed to outdo each other with 256 colors at 640-by-480, then 800-by-600, and 1,024-by-768 pixels. Chip makers added another wrinkle when they developed DAC chips capable of pumping out 16-bit and 24-bit color. IBM tried to reassert itself with the XGA graphics adaptor, but it was too little, too late.

Lacking an IBM standard to mimic, each video card manufacturer was on its own when it came to implementing high- resolution modes. The result was a complete lack of consistency between video cards in mode numbers, extended registers, and drivers. Each manufacturer was responsible for developing drivers for popular programs and providing documentation to software developers. And to be honest, software developers were pretty low priority for most video card makers.

The only glimmer of hope for using a card's high-resolution modes was provided by Microsoft's Windows. As Windows 3.0 and then 3.1 became an industry juggernaut, it forced manufacturers to make Windows video drivers a priority. With a Windows video driver, any Windows program could take full advantage of the video card's enhanced graphic capabilities. The only drawback—Windows was anything but an ideal choice for graphics-intensive games. Microsoft's WinG may bridge that shortfall in the future, but up to this point, games have pretty much been forced to stick with DOS and the VGA standard.

To alleviate this problem, the Video Electronics Standards Association (VESA) was founded in 1989 as a non-profit organization to set and support

Matt Pritchard

industry-wide open standards for the PC environment. Starting in August 1989 with a standard mode number for 800-by-600, 16-color Super VGA graphics, VESA has been busy working with the PCVESA terms standards for video and related items. VESA came up with a system of extensions to the normal VGA BIOS that manufacturers could retrofit to existing cards. In October 1991, the organization released the VESA BIOS Extension (VBE) standard, version 1.2. To keep up with evolving video cards, and the shift to protected mode programs, the VBE 2.0 standard was released in November 1994.

Well, all this history is nice to know, but by now you may be asking "What's in it for me, the game software developer?" The answer is simple: A single, clearcut method for using high-resolution video modes on most of the video cards out there today. The rest of this article will give you an overview of what VESA BIOS extensions can give your software, how they work, and how you go about coding them.

VESA Terms and Features

For starters, we need to clarify our terms. VESA has developed many standards for power management, sound, hardware and I/O bus interfaces and PC Videos BIOSes. This article only covers VESA Video BIOS standards known as Video BIOS Extensions or VBE for short.

A video card can provide Video BIOS extensions in different ways. The ROM BIOS physically on the video card can support the VESA VBE 1.2 or 2.0 features directly. Most new video cards do this. However, due to limited ROM space, some new cards don't have a complete VESA VBE implementation in their ROMS. For older cards, the manufacturer can provide a driver that is placed in the AUTOEXEC.BAT or CONFIG.SYS that augments the card's ROM BIOS and provides the VESA VBE specific functions that reside in RAM. Or a third-party driver, such as SciTech Software's UniVBE program, can provide the VESA VBE either as a TSR or library that can be directly linked into a program, providing VESA VBE services only while that program is running. In many cases, especially with older video cards, some ROM BIOSes and drivers have bugs or are missing features, with hardware panning being the biggest culprit. In these cases, a third party driver usually works better than the manufacturer supplied driver or ROM BIOS.

The major features that VESA VBEs provide are:

- A consistent way to identify a video card, how much memory it has, and what video modes and features it is capable of.
- A consistent way to set high resolution (super VGA) video modes.
- A consistent way to access all of the memory on a video card through a "window" of memory, usually at segment A000.
- A consistent way to save and restore super VGA Modes.
- A consistent way to set logical screen sizes and page-flip Super VGA modes.
- A consistent way to access video DAC registers, including 8-Bit DACs and DACs that are not VGA compatible.

VESA was founded to create industry-wide open standards for the PC environment. By applying these standards, developers have the benefit of a universal method for using high-resolution video modes.

The VESA VBE 2.0 standard added two major features:

- Support for fast direct access to VESA VBE function from protected-mode programs.
- Support for direct access to all video memory in the form of a Linear Frame Buffer, accessible from protected mode.

Accessing VESA VBE Features

All VESA VBE functions and features are accessed through the INT 10h interface. For programmers unfamiliar with this process, allow me to explain. In addition to traditional hardware interrupts, the Intel CPU design allows for software to issue interrupts. Software interrupts are numbered from 0 to 255 and allow a program to call a function without knowing where it is in memory. Interrupt number 10h (16 decimal) is used to access video BIOS services. These services can be in the video card's ROM, DOS, or a driver loaded into RAM and can be chained together.

To call a Video BIOS function, the function's number is put into the CPU's AX register and any parameters are put in the other CPU registers in a manner that is function specific. Then the INT 10h instruction is executed. The CPU transfers control to the interrupt handler which processes the selected function and returns to the program just as if a normal function call had taken place.

The starting place for any VESA-aware program is determining if there are VESA BIOS Extensions available to program. All VBE function numbers have the value 4Fh in the AH register and the actual function number in the AL register (The AH and AL registers make up the AX register).

The VBE function to return Super VGA information is number 0, and before you can call it, you must have a buffer into which VESA information can be put. The VBE 2.0 standard increased the size of the buffer from 256 to 512 bytes, but to keep older VESA programs from breaking you must put the four-character string VBE2 at the start of the buffer to get the VBE

Listing 1. VESACHK.C (Continued on p. 45)

```
/* ============================================================ */
/* VESACHK.C - Check for VBE Extension and report mode available */
/* ============================================================ */

#include <stdio.h>
#include <dos.h>
#include <string.h>

#define uchar unsigned char
#define uint  unsigned int

typedef struct {

    /* These items defined in the VESA VBE 1.2 Spec */

    char            Vbe_Signature[4];       /* "VESA" */
    int             Vbe_Version;            /* Vesa version *
    char far *      Oem_String_Ptr  ;       /* Name of Video Card */
    uchar           Capabilites[4];         /* Bitmapped features */
    uint far *      Video_Mode_Ptr;         /* >List of Video Modes */
    int Total_Memory;                       /* Video Mem / 64K */

    /* These items added in the VESA VBE 2.0 Spec */

    int             Oem_Software_Rev;
    char far *      Oem_Vendor_Name_Ptr;
    char far *      Oem_Product_Name_Ptr;
    char far *      Oem_Product_Rev_Ptr;

    uchar           reserved[222];          /* From the 1.2 spec */
    uchar           Oem_Data[256];          /* VBE 2.0 OEM strings */

} VbeInfoBlock;

int get_VESA_version(VbeInfoBlock far* , int*, int*);

void main ( void )
{

    VbeInfoBlock    VESA_Info;
    int             VESA_Version, MajorVer, MinorVer, Mode;
    uint far *      Mptr;

    VESA_Version = get_VESA_version(&VESA_Info, &MajorVer, &MinorVer);

    if (VESA_Version) {
        printf ("This video card has VBE version %d.%d extensions.\n", MajorVer,MinorVer);
        printf ("The Card Name is ´%Fs´\n",VESA_Info.Oem_String_Ptr);

        /* Display List of Available Modes */

        printf ("\nThe Following Modes are supported:\n");
        Mptr = VESA_Info.Video_Mode_Ptr;
        while ( (Mode = *Mptr++) != 0xFFFF) {
            printf ("mode %4.4xh      ",Mode);
        }
        printf("\n\n");

    } else {
        printf ("This video card does not have VESA BIOS support\n");
    }


}

int get_VESA_version( VbeInfoBlock far* VESA_Info,int * MajorVer, int * MinorVer )
{

    union REGS in_regs, out_regs;
```

Listing 1. VESACHK.C (Continued from p. 44)

```
        struct SREGS seg_regs;
        int   ver, m;

        /* Prepare VESA info Buffer */

        memset( VESA_Info, 0, 512);
        memcpy( VESA_Info->Vbe_Signature, "VBE2", 4);

        /* Call Int 10, VBE Function (4F)00h */

        in_regs.x.ax = 0x4F00;
        seg_regs.es  = FP_SEG(VESA_Info);
        in_regs.x.di = FP_OFF(VESA_Info);

        int86x(0x10, &in_regs, &out_regs, &seg_regs);

        /* Check if VBE Extions present & Get Revision Level */

        if ( (out_regs.x.ax != 0x004F) ||
        (memcmp(VESA_Info->Vbe_Signature, "VESA", 4) != 0) )
        {
            return 0;
        } else {
            ver = VESA_Info->Vbe_Version;
            *MajorVer = (ver >> 8);

            /* bug check - Some BIOSes return 0x0102 instead of 0x0120 */
            /* Check for Minor version in wrong nibble and correct */

            m = (ver & 0xFF);
            if ((m != 0 ) && ((m & 0x0F) != 0)) {
                *MinorVer = m * 10;
            } else {
                *MinorVer = (m >> 4) * 10;
        }
        return ver;
        }
    }
```

2.0 specific additional information. After calling the Video BIOS, if the AX register has the value 004Fh in it, then your BIOS has VESA extensions, otherwise it doesn't. Listing 1 is a function written in Borland C 3.1 that can tell you what version of VESA extension your card supports, if any, and what VESA display modes it has.

Once you know that your video card supports VBE, you still need to find out what video modes your card actually supports. Two pieces of information in the VESA information buffer do this for you. The first is the Total_Memory word, which returns the amount of actual memory on the video card in 64K blocks. The second is the Video_Mode_Pointer, which is a pointer to a list of video modes that the card supports. The mode numbers can include modes that are unique to your video card, numbered from 14h to 7Fh, along with mode numbers that VESA has designated to be standard on all cards, 100h to 11Bh. Because of this, it is possible for a mode to be listed under two different mode numbers, both equally valid. See Table 1 for a list of standard VESA mode numbers.

Now, just because a mode number is listed doesn't mean that the mode is actually supported (for example, there may not be enough video memory). To check out the modes you want to use, you call VBE function 01h, return Super VGA mode information. You must provide a 256-byte buffer for information to be returned in. If AX comes back with anything other than 004Fh then the mode is not available. If AX is 004Fh and the first bit of the first word in the buffer (Mode_Attributes) is 0, then the mode is not available.

The mode information block returned by function 01h contains a wealth of information about how to actually use that mode. That information includes:

- Whether the mode is text or graphics, monochrome or color.
- Whether the BIOS text functions are available in that mode.
- The screen resolution and character cell sizes.
- The memory access model and pixel layout used by that mode.
- The addresses and sizes of the memory regions used to read and write data in that mode.
- Whether or not the VGA DAC can be used in that mode.

VESA Memory Addressing

The whole point of using VESA Video BIOS extensions is so that you can write a program that doesn't have to know a thing about the video card in your system yet can still blast data to the screen at full speed.

Making a VESA VBE-aware program requires some extra work on the part of the programmer, especially in the area of graphics memory addressing and pixel data organization. To make one or more megabytes of video memory accessible, VESA allows for one or two "windows" into video memory of up to 64K to be mapped into the CPU's address space.

In practice, this usually means a 64K window at segment A000, but there is no guarantee—you need to look. It's possible two independent 32K windows might exist at segments A000 and A800. Or you might find two overlapping 64K windows at segment A000—one for reading data from one part of video memory and another for writing data to a different section of video memory.

To access all of video memory VBE function {05h, Display Window Control} lets you change the position in video memory that the window accesses. This is also known as "bank switching." One thing you can count on varying from card to card is the window granularity. Window granularity is the "coarseness" by which the window can be positioned in video memory. Some cards can let you move the window address around in 1K increments while others can only move 32K or 64K at a time. Your drawing code will have to take into account that some cards may need to switch banks in the middle of drawing while others won't. For speed considerations, bank switching is the only operation that you don't need to execute an interrupt to access. The VBE can give your program the address of a direct function to switch banks for the current video mode.

For protected mode programs, VBE version 2.0 opens up a whole new set of possibilities. With the advent of local bus architectures, it became popular among video card manufacturers to support linear frame buffers. A linear frame buffer is where a block of very high memory addresses are directly mapped into the card's video memory, eliminating the need for bank switching entirely. Some cards offer both modes of memory access, others just one. Developers should expect linear frame buffers to become the preferred method of video memory access in the years ahead.

Reading and Writing Video Memory

Once you have gotten through VESA mode selection and memory addressing, actually reading and writing image data is anticlimactic. The most common memory layout for 256-color modes is identical to mode 13h. The only additional need is to use bank switching to move the currently needed portion of video memory into the access window. For high-color and true-color modes, you need to do more work. The size and position of each color component can vary and your drawing code will need to adapt to the different layouts. For example, many new cards use 32-bit pixels for 24-bit true color modes for higher internal performance. Fortunately, the VBE mode information block contains detailed information on the layout of pixels in these modes. Listing 2 demonstrates drawing simple patterns using the VESA 640-by-480 256-color mode.

Panning and Page Flipping

One of the biggest benefits of VBEs for game programs is that it makes possible page-flipping and smooth, high-resolution animation. VBE functions are provided to let the program change the logical width of a scan line and select the position in video memory from which the screen display starts. Just like in Mode X, these features let programs scroll the screen and use double and triple animation buffering.

These functions are simple to use, but there are some catches. Different video cards have different limits on how small an adjustment can be made to line widths and screen start position. For example, a card may only allow the start position to be on a Dword (4-byte) boundary, while another might require paragraph (16-byte) boundaries. In true color modes, the pixel size is usually 3 bytes with no unused bytes. So unless the screen start position is a multiple of 4 pixels (12 bytes), the display would start in the middle of a pixel, scrambling the color data stream! Fortunately, page flipping can avoid this easily by starting each page on a large boundary such as 4K. Smooth panning, on the other hand, suffers badly on some cards.

Palette Functions

VESA VBEs also provide palette functions. Two aspects of the palette functions make them more desirable than the traditional method of program the VGA card directly. The first is that the VGA Palette registers don't work in super VGA modes on some newer cards, while using the VBE palette functions does. The second benefit is that VESA VBEs provide access to 8-bit DACs on cards that have them. Normal VGA DAC registers only have 6 bits of color data per color, but most newer cards can process 8 bits of color data per color. Using 8-Bit DACs allows for smoother looking color ranges and color effects.

Conclusion

A well written VESA-aware program should have few problems running high performance graphics on video cards that have yet to be introduced, including the increasing number of cards that do not have a VGA-compatible core for their high-resolution graphic modes.

If I may share an observation from experience, when making a VESA VBE-aware program for the first time, you

MORE INFO

SciTech Software makes a VESA VBE 2.0 compatible "universal" driver that supports over 160 different graphics chips. Its VBE driver is available in TSR form as shareware for end users or developers. Linkable library versions are also available for software developers. For more information contact SciTech at (916) 894-8400 or via e-mail at sales@scitechsoft.com.

For more information about VBE contact Kevin Giliett at VESA. The VESA VBE 1.2 and 2.0 Specs can also be ftp'ed on the internet from x2ftp.oulu.fi. Write to VESA at 2150 N. First St., Ste. 440, San Jose, Calif. 95131-2029 or call (408) 435-0333.

Table 2. VESA VBE Functions (AH = 4FH)

AL=	Function Description	VBE Revision
000h	Return VBE Controller Information	1.2
01h	Return VBE Mode Information	1.2
02h	Set VBE Mode	1.2
03h	Return current VBE Mode	1.2
04h	Save/Restore VBE Mode State	1.2
05h	Display Memory Window Control	1.2
06h	Set/get Logical Scan Line Length	1.2
07h	Set/Get Display Start	1.2
08h	Set/Get DAC Palette Control	1.2
09h	Set/Get Palette Data	2.0
0Ah	Return VBE Protected-Mode Interface	2.0

Listing 2. VESADEMO.C (Continued on p. 49)

```
/* ============================================================ */
/* VESADEMO.C - A simple 640x480 VESA Graphics Demo             */
/* ============================================================ */

#include <stdio.h>
#include <dos.h>
#include <string.h>

#define uchar unsigned char
#define uint  unsigned int

    /* Stucture to hold VESA VBE Get SVGA Info Results */

typedef struct {

    /* These items defined in the VESA VBE 1.2 Spec */

    char            Vbe_Signature[4];       /* "VESA" */
    int             Vbe_Version;            /* Vesa version */
    char far *      Oem_String_Ptr;         /* Name of Video Card */
    uchar           Capabilites[4];         /* Bitmapped features */
    uint far *      Video_Mode_Ptr;         /* >List of Video Modes */
    int             Total_Memory;           /* Video Mem / 64K */

    /* These items added in the VESA VBE 2.0 Spec */

    int             Oem_Software_Rev;
    char far *      Oem_Vendor_Name_Ptr;
    char far *      Oem_Product_Name_Ptr;
    char far *      Oem_Product_Rev_Ptr;

    uchar           reserved[222];          /* From the 1.2 spec */
    uchar           Oem_Data[256];          /* VBE 2.0 OEM strings */

} VbeInfoBlock;

    /* Stucture to hold VESA VBE Get Mode Info Results */

typedef struct {

    /* This section available is all VESA revisions */

    uint            Mode_Attributes;
    uchar           Win_A_Attributes;       /* Memory Window A */
    uchar           Win_B_Attributes;       /* Memory Window B */
    int             Win_Granularity;        /* in 1K units */
    int             Win_Size;               /* Window Size in K */
    uint            Win_A_Segment;          /* Segment of Window A */
    uint            Win_B_Segment;          /* Segment of Window B */
    void far *      Win_Func_Ptr;           /* Bank Switch func addr */
    int             Bytes_Per_Scan_Line;    /* just what it says */

    /* This section available is VBE revision 1.2 & up */

    int             X_Resolution;
    int             Y_Resolution;
    char            X_Char_Size;
    char            Y_Char_Size;
    char            Number_Of_Planes;
    char            Bits_Per_Pixel;
    char            Number_Of_Banks;
    char            Memory_Model;
    char            Bank_Size;
    char            Number_Of_Image_Pages;
    char            Reserved1;

    /* This section is used for Memory_Model 0x06 and 0x07 i.e. Direct
     Color and YUV color modes */

    char            Red_Mask_Size;
    char            Red_Field_Position;
    char            Green_Mask_Size;
    char            Green_Field_Position;
    char            Blue_Mask_Size;
    char            Blue_Field_Position;
    char            Rsvd_Mask_Size;
    char            Rsvd_Field_Position;
    char            Direct_Color_mode_Info;

    /* This section available is VBE revision 2.0 & up */

    void far *      Physical_Base_Ptr;
    void far *      Off_Screen_Mem_Offset;
    int             Off_Screen_Mem_size;

    char            Reserved[206];

} ModeInfoBlock;

    /* Global Registers for calling INT 10h */

union REGS in_regs, out_regs;
struct SREGS seg_regs;

    /* Function Prototypes */

int get_VESA_version(VbeInfoBlock far* , int*, int*);
int check_VESA_Mode(int, ModeInfoBlock far* );
int set_VESA_Mode(int);

void main ( void )
{

    VbeInfoBlock    VESA_Info;
    ModeInfoBlock   MODE_Info;

    int             VESA_Version, MajorVer, MinorVer;

    /* Check if VESA Video BIOS Extension installed */

    if (!(VESA_Version = get_VESA_version(&VESA_Info,
        &MajorVer, &MinorVer))) {
        printf ("Unabale to run demo.\n");
        printf ("This video card does not have VESA BIOS support\n");
        return;
    }

    /* Check if Mode 101h is available */

    if (!check_VESA_Mode(0x0101, &MODE_Info)) {
        printf ("Odd... This card has VESA BIOS extension but no "
                "support for mode 101h\n");
        return;
    }

    /* Set display to VESA Mode 101h */

    set_VESA_Mode(0x0101);

    /* Draw our Test pattern and wait for a keypress */

    draw_VESA_triangle_pattern(&MODE_Info,640,480);

    /* restore text Mode */

    set_VESA_Mode (0x03);
    printf ("this VESA 640x480x256 demo is done.\n");
```

Listing 2. VESADEMO.C (Continued from p. 48)

```
    return;
}

    /* Function to get VESA VBE Mode Info for a given mode */

int check_VESA_Mode(int Mode_Num, ModeInfoBlock far* MODE_Info)
{

    /* First we get info on the desired mode */

    memset( MODE_Info, 0, 256);       /* Clear Mode Info */

    in_regs.x.ax = 0x4F01;            /* Get mode Info Function */
    in_regs.x.cx = Mode_Num;          /* Mode number */
    seg_regs.es  = FP_SEG(VESA_Info); /* Buffer to hold results */
    in_regs.x.di = FP_OFF(VESA_Info);

    int86x(0x10, &in_regs, &out_regs, &seg_regs);

    if (out_regs.x.ax != 0x004F) {    /* Did an Error occur? */
        return 0;
    } else {
        return Mode_Num;
    }

}

    /* Function to perform VESA VBE Mode Set */

int set_VESA_Mode(int Mode_Num)
{

    in_regs.x.ax = 0x4F02;            /* Get mode Info Function */
    in_regs.x.bx = Mode_Num;          /* Mode number */

    int86x(0x10, &in_regs, &out_regs, &seg_regs);

    if (out_regs.x.ax != 0x004F) {    /* Did an Error occur? */
        printf ("Error attempting to set Mode %xh\n", Mode_Num);
        return 0;
    } else {
        return -1;
    }

}

int get_VESA_version( VbeInfoBlock far* VESA_Info,
                        int * MajorVer, int * MinorVer )
{

    int   ver, m;

    /* Prepare VESA info Buffer */

    memset( VESA_Info, 0, 512);
    memcpy( VESA_Info->Vbe_Signature, "VBE2", 4);

    /* Call Int 10, VBE Function (4F)00h */

    in_regs.x.ax = 0x4F00;
    seg_regs.es  = FP_SEG(VESA_Info);
    in_regs.x.di = FP_OFF(VESA_Info);

    int86x(0x10, &in_regs, &out_regs, &seg_regs);

    /* Check if VBE Extions present & Get Revision Level */

    if ( (out_regs.x.ax != 0x004F) ||
        (memcmp(VESA_Info->Vbe_Signature,
                 "VESA", 4) != 0) )
        {
            return 0;
        } else {
            ver = VESA_Info->Vbe_Version;
            *MajorVer = (ver >> 8);

            /* bug check - Some BIOSes return
              0x0102 instead of 0x0120 */
            /* Check for Minor version in
              wrong nibble and correct */

            m = (ver & 0xFF);
            if ((m != 0 ) && ((m & 0x0F) != 0))
            {
                *MinorVer = m * 10;
            } else {
                *MinorVer = (m >> 4) * 10;
            }
                return ver;
        }
    }
```

should start by getting the VESA VBE 1.2 and 2.0 specifications and studying them. Then I heartily recommend that you obtain other VESA VBE code and study it. VBE support varies and the specifications leave some room for interpretation. For example: Where the VBE 2.0 says “All other registers are preserved,” many VBEs still trash the high words of extended registers. Another example is that some VBEs return a value of 0 for a window granularity of 64K instead of returning 64.

There is a wealth of information and detail involved in using VESA Video BIOS Extensions, and we didn't have room in one article to cover it all. I do hope you now have an idea of what you'll be facing when you develop a program that uses VESA graphic modes. Making the effort to use VESA VBEs can bring great rewards to the developers who want their programs to stay on the cutting edge. Until next time, happy coding! ■

Matt Pritchard is presently launching his software company Innovatix in Garland, Texas, and is the author of MODEX105, a comprehensive freeware ModeX library. You can reach him via e-mail at matthewp@netcom.com or through Game Developer *magazine.*

Back

Wing Commander III

Wayne Sikes

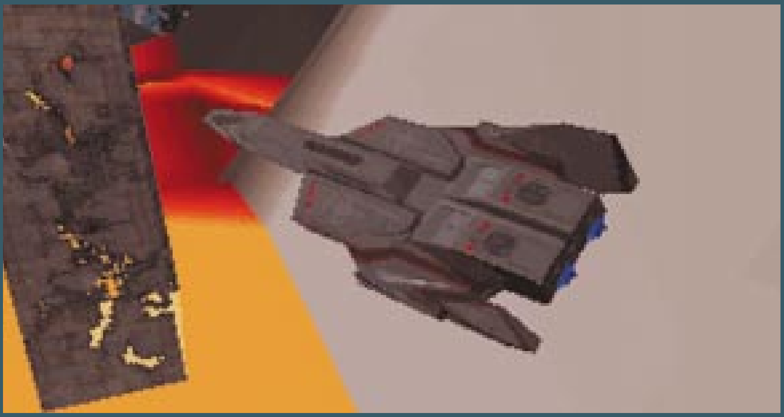
There are a lot of fighter pilot games out there these days. Wayne Sikes navigates through Origin's Wing Commander III, the third release of the Wing Commander series, and it measures up impressively.

In this edition of the Chopping Block, I review Wing Commander III by Origin Systems Inc. This technical review contains minor spoilers. Wing Commander III is the third installment in the Wing Commander series. This version is far more technologically advanced than previous Wing Commander episodes. Game features include SVGA digitized movies using well-known actors at the game intro and end as well as in all cutscenes; the graphic engine uses polygon-based, texture-mapped VGA and SVGA graphics, which result in impressive, realistic looking spacecraft; the user interface has a clean simple layout and is easy to use; and the General MIDI soundtrack and digitized sound effects add a full layer of realism and excitement to the game. Wing Commander III appears to use a combination of Origin's trademarked RealSpace technology—which was used in other Origin games such as Privateer, Strike Commander, and Pacific Strike—mixed with advanced polygonal routines created specifically for Wing Commander III.

This Is A Big Game!

The combined English, French, and German versions of Wing Commander III comes on four CD-ROMs. Total distribution size is about 2.3 gigabytes. The Wing Commander III executable (WC3.EXE) plus several primary data files are duplicated on each CD-ROM; the unique data files require over 1.8 gigabytes of CD-ROM storage space. The movie files are by far the largest files in the game and range in size from 108MB to 408MB.

Installing Wing Commander III is quite an extensive process. To minimize data file loading times during game play, a full installation is required


The latest addition to the Wing Commander series features more technologically advanced graphics that allow for impressive, realistic looking spacecraft.

Listing 1. General TRE File Format

```
FILE          DATA
OFFSET        SIZE          DESCRIPTION

0-3           char          "XTRE" char string.
4-7           char          Blanks. 4 bytes of 00.
8-11          long          Starting Offset of the Indirect Record Pointer Table.
12-15         long          Starting Offset of the Path Name Table.
16-19         long          Starting Offset of the Record Pointer Table.
20-23         long          Offset of the first Data Record in this TRE file.
24-(xx-1)     var           Indirect Record Pointer Table.  This table contains indirect
                            pointers, or "pointers to pointers".  The values in this table
                            point into the Path Name Table or into the Record Pointer
                            Table.
xx-(yy-1)     var           Path Name Table.  Table of variable-length records containing
                            system path names and other path-related data.

yy-(zz-1)     var           Record Pointer Table. This table consists of structures
                            containing the file offsets and sizes of the Data Records
                            in this TRE file.

zz-EOF        var           Individual Data Records.

NOTES:
1.  var refers to data having variable sizes.
2.  xx is the Starting Offset of the Path Name Table.
3.  yy is the Starting Offset of the Record Pointer Table.
4.  zz is the Offset of the First Data Record.
5.  EOF refers to the end of the file.
```

and uses about 40MB of hard disk space. A VESA driver is required for SVGA graphics. Because Wing Commander III is such a large game and weighs heavily on system resources, it is no surprise that many users have had trouble installing it. (I experienced installation troubles because the size of my CD-ROM drivers combined with a Pro Audio Spectrum driver exceeded the minimum conventional memory requirements for my 8MB RAM system. The game features a boot disk creation routine that solved my memory management problems.)

Wing Commander III requires a minimum of 360K conventional RAM plus 7,000K of either extended or expanded memory. Access to memory above 640K is provided by Origin's custom JEMM.OVL protected mode (DPMI), 32-bit driver. Virtual memory is emulated through the use of a swap file named SWAPFILE.$$$. The size of the swap file is generally determined by the amount of free hard disk space, and it is usually 15MB to 20MB. The swap file is created when Wing Commander III is booted and erased when you exit the game. (This file is erased when you exit the game via the Alt-X keystrokes. So be sure to erase the swap file from your Wing Commander III hard drive subdirectory if you exit the game using any other method such as rebooting your PC.)

The primary Wing Commander III executable, WC3.EXE, is about 1.1MB in size. Wing Commander III was developed using the MetaWare High C/C++ system and is written primarily in C++. (The MetaWare High C system features C++ templates and a compiler for developing 32-bit protected mode applications.) The WC3.EXE game engine includes a built-in Examination Mode that provides general development and debugging information while the game is running. Developers can observe the status of the numerous game engine flags, accumulators, stack manipulation calls, and game flow data such as act, scene, and mission information. I found several undocumented keystrokes while exploring game operation, and I'll summarize them further on.

Data Storage

Essentially all of the game data is stored in files suffixed with TRE. (I will refer to these data files as TRE files.) TRE files use a very flexible, open-ended data storage format that allows for very large files. You can store hundreds of megabytes of data in a single TRE file.

Listing 1 gives a general summary of the TRE file format. The first four bytes in the file contain the XTRE ASCII text string. The XTRE string possibly indicates this file contains cross-referenced data records. The bytes at offsets 8 to 23 contain four pointers. The first three pointers give the file offsets (offsets into the TRE file with the first byte in the file referenced as byte 0) of three data tables. The fourth pointer gives the offset of the first Data Record in the file.

The first data table in the TRE file, the Indirect Record Pointer Table, contains pointers (file offset values) that reference data in the other TRE file data tables. In C nomenclature, the Indirect Record Pointer Table contains pointers to pointers. The Indirect Record Pointer Table entries "point" to various "pointers" contained in other TRE file tables. Each entry in this table is an 8-byte structure. The first four bytes are bit flags and the last four bytes contain the pointer expressed as a 32-bit long value. I observed that the Indirect Record Pointer Tables of several TRE files had "blanks" or empty data slots. As I mentioned earlier, the TRE file format is flexible and expandable, and these empty data slots are probably part of this expandable format.

The Path Name Table follows the Indirect Record Pointer Table. Embedded in this table are the ASCII text strings of the source paths along with binary (possibly file offset) data related to each path name.

The Record Pointer Table follows the Path Name table. The Record Pointer Table contains the file offsets and sizes of the Data Records stored in the TRE file. The offset and size of each Data Record is contained in an eight-byte structure. The first four bytes of each structure give the offset of the Data Record as a long value and the last four bytes contain the long size.

The individual Data Records follow these tables. These Data Records vary in size from just a few bytes to several thousand bytes and contain the actual game data—fonts, palettes, space and vehicle graphics, mission data, game flow data, artificial intelligence information, and so on. Some of the game data is stored in IFF format, which I'll cover in the next section.

Does the TRE file structure seem a bit confusing? Although somewhat difficult to understand, this file format is cleanly organized. Basically, the entries in the three data tables at the top of a TRE file reference—or point to—all the Data Records found in the TRE file. In other words, these tables tell the game engine where to find the desired data.

As an example, let's trace the path the game engine follows to find a look-up table containing the file names of the Wing Commander III missions. The game engine opens the GAMEFLOW.TRE file and reads down the list of entries in the Indirect Record Pointer Table until it finds the entry corresponding to our mission file name look-up table. The engine reads the hex values `62 2b 01 3f  84 07 00 00`. As the Indirect Record Pointer Table entries showed us, the `62 2b 01 3f` values are bit flags and the `84 07 00 00` bytes contain the long pointer.

The data beginning at hex offset `784` (don't forget about the PC's method of storing data in low byte and high byte format) references an entry in the Record Pointer Table that reads as ae `13 f4 00 ac 04 00 00`. Keeping in mind that the Record Pointer Table contains the file offset and size of the actual Data Record, you'll see the data for the mission file names is stored at hex offset `f413ae` and is `4ac (hex)` bytes in size. If you verify my example by looking in the GAMEFLOW.TRE file for the mission file names, you will see these names (as ASCII text) listed using Origin's IFF file format. This format is discussed in the next section. Listing 2 gives a summary of the Wing Commander III mission file names that we've extracted from the look-up table.

IFF File Formats

Origin has used the IFF file format in several games including Privateer, Strike Commander, and Pacific Strike (see "Bandits at 0x1200 High!" Chopping Block, Dec. 1994, for a discussion of the IFF files used in Pacific Strike). To summarize the IFF file format, an IFF file is composed of one or more forms, with each form containing one or more records. The following general rules are used in IFF file formatting:

- All forms have a header consisting of the "form" text string followed by a 4-byte number. This number is the number of data bytes in the form.
- All records have a header consisting of a record name (that can be up to eight bytes of text characters) followed by a 4-byte number. This number is the number of data bytes in the record.
- Records can be located both inside and outside of a form.

Mission Ordering

Refer to Listing 2 and examine the hex code that include the mission file name look-up table we discussed previously, and you will notice that each mission is referenced by a unique record name and data group. Each mission record name is a text number and the data is a text file name. For example, mission `0014` is stored in `file misnd003`.

The first mission's record name is `LOOK0000` (file name `misna001`) and the last mission's record name is `0066` (`misnp000`). The missions are generally grouped according to the space system in which they are located. For example, missions `0036` (`misnl001`) through `0039` (`misnl004`) occur in the Alcor System.

Even though the missions have record names giving a one-up order, the game engine does not play the missions in the specified order. The order in which the missions are played is determined by how well or how poorly the

Listing 2. The Mission Look-Up Form

RECORD NAME	RECORD DATA (MISSION FILE NAMES)
LOOK0000 0001 0002 0003	misna001 misna002 misna003 misna004
0004 0005 0006 0007	misnb001 misnb002 misnb003 misnb004
0008 0009 0010 0011	misnc001 misnc002 misnc003 misnc004
0012 0013 0014	misnd001 misnd002 misnd003
0015 0016 0017	misne001 misne002 misne003
0018 0019 0020	misnf001 misnf002 misnf003
0021 0022 0023	misng001 misng002 misng003
0024 0025 0026	misnh001 misnh002 misnh003
0027 0028 0029	misni001 misni002 misni003
0030 0031 0032	misnj001 misnj002 misnj003
0033 0034 0035	misnk001 misnk002 misnk003
0036 0037 0038 0039	misnl001 misnl002 misnl003 misnl004
0040 0041 0042	misnm001 misnm002 misnm003
0043 0044 0045	misnn001 misnn002 misnn003
0046 0047 0048	misnp001 misnp002 misnp003
0049 0050	misnr001 misnr002
0051	misnb2ne
0052	misnc2ne
0053	misnd3bd
0054	misnd1b
0055 0056 0057 0058 0059 0060	tsim001 tsim002 tsim003 tsim004 tsim005 tsim006
0061	misnl01d
0062 0063 0064 0065	tsim007 tsim008 tsim009 tsim010
0066	misnp000

player is performing. For example, assume you, the player, are doing poorly and you fail to complete mission 0039 (misnl004). (Mission 0039 requires that you fly down to Alcor V, take out all the ground installations, and rescue the scientist who knows how to make the Tembler Bomb.) This failure sends you to the Proxima System to play missions 0049 (misnr001) and 0050 (misnr002). (Mission 0050 is the horrid mission where never-ending waves of enemy Kilrathi appear; the game never ends and you have lost.)

In a different scenario, if you are doing well and complete mission 0039 successfully, you jump to the Freya System and begin mission 0040 (misnm001). If all goes well after a few missions, you will be heading to downtown Kilrah with the Tembler Bomb during mission 0048 (misnp003).

Note that more than one mission file can be played during a mission. For example, consider mission 0041 (misnm002). The misnm002 file is played during the initial space combat part of the mission. Next, the misnm02g file plays when the player is doing combat on the surface of a planet. Lastly, file misnm02b plays during space combat at the end of the mission.

Now that I have explained IFF file formatting and how the mission file names are stored, I have a little homework for you. Try to find the look-up table that contains the names of the space systems. Hint: The space systems IFF file data is in the GAMEFLOW.TRE file. What is the record data that corresponds to the record name LOOK0000 in the space systems look-up table?

Undocumented Keystrokes

I found several undocumented keystrokes that are not referenced in the game's literature. These keystrokes only appear to be active during the flight mode (when you are in the cockpit flying a mission). The Alt-V keystroke displays the game version and the file name of the current mission. The displayed file name is found in the mission file name look-up table previously discussed. The Ctrl-F keystroke toggles the readout of the current frame rate in frames per second. The Ctrl-M, Ctrl-J, and Ctrl-K keystrokes select the mouse, joystick, or the keyboard, respectively, as your primary flight control device.

Make Your Missions A Little Easier (Or Harder)

While playing through the various missions and scenarios I quickly decided that my computer pilot needed help surviving some of the missions. Altering the game weapons (guns and missiles) seemed to be the best method for increasing my pilot's survival time. I wrote a utility called WCEASY that edits most of the primary gun and missile parameters. The WCEASY.ZIP routine can be found on CompuServe in the Flight Simulation Forum (GO FSFORUM), Space Combat Library.

If you want to write your own utilities for altering Wing Commander III game play, let me warn you that the Wing Commander III game engine is very sensitive and very inflexible. The complexity of this engine is such that little allowance is made for altering weapons or missions in drastic ways. (I wrote an editor for Privateer, PREDIT, a couple of years ago that let you add several hundred missiles to your vehicle. Do not do this with Wing Commander III!)

Also some of the mission (and possibly other nonmission) data is encrypted or somehow abbreviated in structure—possibly a new type of IFF file format. Unless you study this format carefully, I would not advise changing it. Any unallowed changes will at best cause the game to halt and will quite possibly crash your PC. (Don't forget to erase the Wing Commander III swap file on your hard drive if your PC crashes.)

Wing Commander III: The Next Generation

The first two installments of Wing Commander were impressive to say the least. Wing Commander III is destined to be a landmark that other gaming companies will try to reach in the foreseeable future. The beauty of its graphic renderings combined with a good, well-acted storyline combine to create a classic game. I can honestly say that very few games hold my attention long enough to keep me playing and replaying scenarios. I found myself constantly wanting the current mission to be completed just so I could see what was going to happen next. The term "Interactive Movie" doesn't seem to do justice to what Origin has created in Wing Commander III.

Wing Commander III exacts a high price in terms of system horsepower. A Pentium is not required, but in order to get the most from the game you will need one. Testing the flight combat sequences using VGA graphics on my 486DX2/50 gave frame rates of three-to-six frames per second during periods of heavy combat action, and the game occasionally hesitated for several seconds in order to load the graphics and sounds for explosions or new waves of enemy vehicles. The loading times ranged from 10 seconds during mission waypoint sequencing to 70 seconds at mission start up. Is a game with longer loading times and lower frame rates worth playing until I can upgrade to a Pentium? If the game is Wing Commander III, you bet it is! ■

Wayne Sikes has been a computer hardware and software engineer for the last 12 years. He has an extensive background in C, C++, and assembly language programming. He also has several years experience as a computer systems intelligence analyst, where he specialized in deciphering and disassembling computer code while working on classified government projects. He has written game engines as well as numerous computer gaming help utilities. You can reach him via e-mail at 70733.1562@compuserve.com or through Game Developer.

Back

Descending to the Top

Alexander Antoniades

What does it take to start your own studio and develop a new game? In 1993, Matt Toschlog and Mike Kulas learned the ropes when they formed Parallax studios. This is their story.

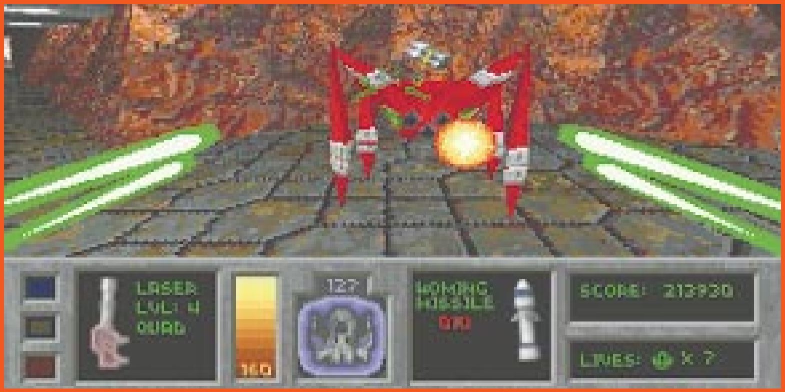
Embarking on an ambitious project such as a commercial game can be a taxing experience, and it's good to have friends to help get you through lean times. For two programmers who had a dream of starting a company and writing their own three-dimensional game, the journey from concept to product became a real learning experience.

Matt Toschlog first met Mike Kulas in 1986 at SubLogic, where they were both working on Flight Simulator 2 for 68000 series processors (Commodore Amiga, Atari ST, and Macintosh). In 1988, Toschlog left SubLogic for Looking Glass, where he worked with Ned Lerner on the Car and Driver computer game. Kulas joined them in 1990 creating game tools for Car and Driver. While they enjoyed working at Looking Glass both Toschlog and Kulas yearned for something different.

Startup

Toschlog and Kulas had each worked on a number of games for various companies and were interested in working on their own creation. Originally, they had kicked around the idea of doing an indoor flight simulator using shaded polygons, similar to the way they'd worked on Flight Simulator 2. But after working on Ultima Underworld they realized that they could use texture mapping to make the game more spectacular. After considering this for a while, they finally got together in April 1993 and wrote the two-page sketch for Descent.

Their next step was to break away and form a company. They settled on the name Parallax studios and started looking for a publisher. One of the first companies they contacted was Apogee. Scott Miller, the president of Apogee, was excited by their proposal and set them up with a contract. Over the next nine months Parallax and Apogee



Descent, the first game developed jointly by Matt Toschlog's and Mike Kulas's Parallax studio, takes place in a mine on a distant planet.

worked closely in putting Descent together. Apogee supplied them with money and experience the company had gained making three-dimensional shareware games, while Parallax implemented any artistic and structural changes that Apogee requested.

While the relationship was workable Apogee had to break it off. Apogee was in the midst of starting its three-dimensional retail project (see “The Sultans of Shareware Go Retail,” By Design, Feb./Mar. 1995), and Descent was entering a more expensive stage of production. Soon Apogee became overextended—it simply had too many games in the works—and made a clean break with Parallax.

So it came to be that nearly a year later, in January 1994, Parallax Software's five-person team (Toschlog, Kulas, artist Adam Pletcher, programmer John Slagel, and level designer Che-Yuan Wang) was almost as far from releasing its first game as when it started. They had serious work to do. Working from the code they *had* finished, they spent the next three weeks frantically putting a mock game to show prospective investors. They sent the mock up, with accompanying letters, to 50 game companies. From the 50 letters, they received three replies, which turned out to be two more than they needed.

Interplay was one of the three interested. The company signed Parallax up immediately. Interplay had been eager to test the waters of the shareware market ever since it had handled the Macintosh release of Wolfenstein 3D. Under the guiding hand of producer Rusty Buchert, Parallax added two more people to their team and ended up finishing Descent eleven months after starting with Interplay.

Descent

One of the aspects that made Descent unique was that it combined all six degrees of freedom from a flight simulator with the traditional Doom texture mapping. Toschlog, Kulas, and their team did this by placing the player in a spaceship surrounded by robot adversaries. This let them create all the moving objects using big polygons that could be texture mapped and viewed from any angle. The game takes place in a mine on a distant planet, so there's no sky to view, and all of the remaining objects that aren't polygons are either rotated, so they're always facing the viewer, or situated in areas where they can't be viewed from above.

Early in the development process, Parallax toyed with the idea of having irregular objects that were flat bitmaps rapidly shift to simulate different viewpoints. But having over 70 bitmaps for each stage of a movement ended up using too much memory, hence the decision to stick with simple polygons.

Even though Descent's game play didn't change much from its beginnings with Apogee, Toschlog and Kulas learned more about project management this time around. For example, Toschlog observed that in a project of this size almost all things need to be done onsite. Sometimes they learned this the hard way, as when the music files came back towards the end of the project too large to fit on two disks. A more positive surprise development was that team members like Adam Pletcher, who was hired as an artist, would often make great suggestions in areas outside of their specialty, such as game design.

Following the lead of many other companies, Parallax used Doom as a benchmark of what to include in its game. While this took off the pressure of worrying about things like cut scenes, the team balked at including one of Doom's most popular features—Network play. Parallax was reluctant to add networking capability because the work required was difficult, but the team also knew it was important for comparison with Doom, so they gave in. Interestingly, it was not network bandwidth that instituted the eight-player limit in the final version of Descent, but the incredible slowdown that occurred if more than eight players were in the same room during a network game.

One area where Parallax took a definitive Doom approach was making Descent a shareware release. This was a decision made during the Apogee days and Interplay decided to stick with it. Despite the fact that Descent was no longer being published by Apogee, Apogee let Parallax post Descent on its highly accessible Software Creations BBS for distribution.

Despite careful planning and programming there were some problems with the 1.0 release of Descent. One bug that bit Parallax hard was when robots recharged their shields after the player died, forcing the player back to the beginning of the level. This took a terrible toll on level 7 (the last level of the shareware version), in which wearing down the “boss” robot's shield over the course of many players' ships was the strategy. “People aren't going to register a game, if they can't even finish the shareware levels,” lamented Kulas.

After fixing the rejuvenated robots bug, another aspect of gameplay was criticized by Descent players. While players could save pilots and return to levels they had completed, there was no in-level save game feature (as in LucasArts's Dark Forces). Convinced that this was too popular a feature to overlooked, Parallax added an in-level save game option to one of its first revisions.

Ascent

What does the future hold for Parallax? First will be the CD-Enhanced version of Descent, which will be to Descent what Doom II is to Doom, that is, more levels with a few new features. After that maybe Descent II, but definitely Descent for Macintosh, for Sega's Saturn, and Sony's Playstation.

While the saying around Parallax during the making of Descent was “This project is our company,” in the future Parallax will concentrate on two projects instead of one. “We can't compete with Wing Commander III,” admits Toschlog, but with two small teams of programmers Parallax hopes its games will be written and designed well enough that it won't have to. ■

Alexander Antoniades is Game Developer*'s editor-at-large. Contact him via e-mail at sander@mfi.com or through* Game Developer *magazine.*

The Three Fs of 3D: Features, Functionality, And 'Fordability

The incredible can be made to look believable with the wide array of special effects available to 3D Studio users. Shown, a scene from The Daedelus Encounter, a Mechadeus title.

Three-dimensional design is hot, and you want it, but a high-end workstation isn't on your shopping list just yet. Where does that leave you? Well, hardware improvements are making the PC a more and more viable alternative, and PC-based three-dimensional modeling, rendering, and animation software is cropping up everywhere to capitalize on that fact. These products all promise the moon, and the ads look swell, but do they really deliver? And how usable are they? This time around I'll give you an artist's view of three packages representing a broad range of features, functionality, and price.

Working in the three-dimensional arena is compute-intensive and on the PC that can turn into a major time-sink, with scene renders dragging on interminably while your poor little hard drive emits soul-wrenching little-engine-that-could sound effects. Fortunately, faster CPUs and system buses, built-in floating-point units, and a new generation of graphics accelerator cards have greatly speeded the PC's ability to deal with the rigors of working in and rendering three dimensions. While of course you want all the processor speed and RAM you can get your little meat hooks into, fledgling three-dimensional artists on a tight budget can squeak by on a 386 with math coprocessor and 8MB RAM. Useful libraries of ready-made objects and textures are generally going to be on CD, so it would be a good idea for even a minimalist setup to include a CD-ROM drive.

Autodesk 3D Studio

Estimates show that 3D Studio now accounts for more than half of the PC modeling and animation market, which makes it a natural standard for comparison. This magazine ran a more in-depth overview of 3D Studio in a previous article ("What is 3D Studio?" June, 1994), but since that time Autodesk has come out with Release 4, and its new features make a revisit worthwhile.

3D Studio runs in DOS, which may seem something of an anachronism these days. While the interface might initially come across as clunky to Windows-dependent users, given the program's depth it's probably much more usable than if it were icon driven. Five modules make up the working area of 3D Studio:

- The 3D Editor, which allows the user to create and alter objects and to position virtual lights and cameras
- The 2D Shaper for creating spline polygons, which then are used to build three-dimensional objects or to define complex paths for animations
- The 3D Lofter, where two-dimensional polygons are built up into objects
- The Materials Editor, to create and edit surface materials
- The Keyframer, for animating lights, cameras, and action.

The user moves back and forth freely within these modules, which are like task-specific workspaces. Release 4 features a significantly speedier 3D Editor, which is where users spend much of their time.

One great advantage of 3D Studio is its support of network rendering. Using a single copy of the software you can add any number of networked computers to a queue to participate in rendering a project, essentially multiplying your computer's resources manyfold and saving you valuable time. Multiple files can be lined up for unattended network rendering while 3D Studio manages the workload, assigning tasks to slave machines as they become available.

A 250-page manual is devoted to outlining the many new features in Release 4. One highlight is the Camera Control and Perspective Match capability, which aids in camera placement and rendering for coordinating your view with a scanned photo image. Another is a Keyframe Scripting Language that uses BASIC-like commands to build in collision detection and elementary physics. There's also a Fast Preview feature that allows rendering modes from wireframe to Phong-highlighted Gouraud shading for full-color previews of animations. Perhaps the coolest new feature is the addition of Inverse Kinematics. This makes possible hierarchically linked object chains—such as jointed limbs—that recognize range-of-movement constraints.

While it is tempting to describe 3D Studio as fully featured, its tremendous flexibility is further enhanced by the availability of myriad third-party plug-ins: alluring little extras that let you, for example, twist, crumple, or explode your three-dimensional creations or add smoke, snow, or fireworks to a scene and much, much more. Autodesk encourages developers to build around its product and claims that new plug-ins are coming at a rate of about one a week—there are already over 200 available. All of which makes for a very richly featured tool, indeed. And it can be yours for about $3,000 (third-party plug-ins, of course, are sold separately).

But what's it like to use? While the breadth and depth of 3D Studio is a spur to the imagination, the sheer scope of it is also somewhat daunting. I found the five manuals (that's right, five) were clearly written and well organized. They include an excellent book of tutorials that's over 500 pages long yet still barely scratches the surface. Savvy users also recommend *Inside 3D Studio* from New Riders Publishing (Indianapolis, Ind.), a hefty book of tips for newcomers and

David Sieks

You need to weigh many factors when choosing software for three-dimensional design. Sure, it's great if a tool is hip and cool—but is it affordable, usable, and versatile?

techniques for professionals, loaded with hands-on examples. These and a lot of patient experimentation will get you going, as the interface—while not necessarily intuitive—is certainly fairly logical once you're grounded. But keep those manuals handy.

New users will quickly find that complex paths for animating objects and cameras can be achieved fairly easily. Geometric forms can likewise be created, deformed, and combined with relative ease to represent an armchair, say, or a spacecraft. Even experienced users, however, complain of the difficulty of creating biomorphic forms. One workaround is to check out plug-ins from Crestline Software (Crestline, Calif.), which offer virtual mannequins with walk and run motion, facial expressions and hand gestures. Crestline's Tim Wilson says models of other creatures, starting with dinosaurs, are in the works.

Joel Gwynn, applications specialist at Cambridge, Mass.-based Designers' CADD Company, an Autodesk reseller and design shop, finds that most 3D Studio purchasers have a specific task in mind when they begin. Designer's CADD can go onsite and train users through that project so the knowledge they're gaining is immediately applicable, which is a great way for a new user to get up to speed. Check with your Autodesk dealer about the availability of training. There are also training seminars from Autodesk University. If you don't have the luxury of a long stretch for innocent, childlike exploration and experimentation, that may be your best bet.

Caligari trueSpace

Or maybe you'll want to take a look at what Caligari Corp. has to offer. "Usable" is the word Caligari prefers in reference to its product, trueSpace for Windows. Rather than splitting functions between various modules its so-called "natural user interface" consists of one large work area. Features are contained on a single menu bar of icons, with additional controls accessible via pull-down menus. Objects are manipulated onscreen in real time. All this is geared toward making trueSpace more readily understandable on an immediate, visual level.

And it is. Navigation around the workspace could not be more straightforward. You can shift your viewpoint up, down, and around in relation to the scene or to a selected object, or along the Cartesian axes. Objects, lights, and cameras are all manipulated in the same way. You can open smaller, auxiliary view windows (and drag them to any location on the screen) to display a camera's viewpoint or to show an almost instantaneous preview of an animation or rendered scene.

I have only one gripe about the interface, and it's admittedly a niggling one. The icon buttons are small and the icons themselves rather obscure. As your mouse pointer encounters each icon, a brief description does appear in the status bar at the bottom of the screen. However, that means you have to keep looking back and forth from the toolbar to the status bar until you become familiar enough with the location and function of each button. That comes with time, but it's a little frustrating for the novice user, especially since so much else about the program is almost immediately comfortable. While I'm griping, let me just add my hopes that in future releases Caligari pays more attention to documentation.

While trueSpace does not support networkable functions, many users speak favorably of its speedy rendering time. Robert Riter of Megabyte Industries (Moravia, N.Y.) used trueSpace to create three-dimensional animations for the Dark Force Productions title Wormhole and is using it throughout a first-person action game of his own currently in development. A former 3D Studio user, his experience is that rendering tasks that might have run overnight with that program can be completed in a fraction of the time with trueSpace.

He also finds trueSpace a far better tool for character creation. Indeed, its modeling tools are not only powerful, flexible, and easy to use, they're getting better—trueSpace for Windows 2.0 was in beta test when I wrote this, with release expected for the summer of 1995. I've been working with the beta myself for the past week and have been impressed with the fluidity of the modeling process. Deformation tools have been improved over version 1.0 so that, using the mouse, you can easily push

Some users find the trueSpace interface lends itself more readily to character creation than do other three-dimensional modelers. Here, a fighter wielding a 'body-ripper' from a game-in-progress by Megabyte Industries.

and pull object faces to alter form. Caligari has also added three-dimensional Boolean operations, which means that you can sculpt complex shapes as though with a chisel, or by fusing three-dimensional shapes together like lumps of clay.

Another coup for Caligari is the addition of real-time solid rendering. If you've used a three-dimensional program before to create anything more than a simple geometric form, you know how confusing it can get at times to work in wireframe. With trueSpace 2.0, it is possible to work with rendered solid forms onscreen in real-time. Two renderers are included: 3DR, from Intel, and Renderware, from Criterion (wireframe is still also an option and remains most useful for point editing). I've found that 3DR updates the screen much more quickly than does Renderware, though the colors tend to get lurid. You can expect even better results if you're using one of the new graphics cards, such as the Matrox Impression, with built-in support for these APIs.

Some further improvements to enhance the program's creation of photorealistic effects are the addition of adjustable depth of field; Adobe Photoshop plug-in support; motion blur; and new seamless procedural textures—including wood, granite, and marble—that allow user-definable settings for roughness, scale, and color. trueSpace 2.0 will retain the $795 SRP and, if it's still not out by the time this hits the stands, you can buy version 1.0 and get a free upgrade from Caligari when the new version does come to market.

Visual Software's Visual Reality

Visual Software (Woodland Hills, Calif.) wants to maketh you like unto a god. Yea, verily, and for cheap, too. Their Visual Reality suite is a virtual Swiss Army knife for the aspiring world maker on a budget. It doesn't attempt to do everything, mind, and it's not exactly a quick learn, but what it does it does thoroughly—and for under $300.

Visual Reality consists of five separate but cooperative applications:

- Renderize Live is the workhorse of the group. It serves as the stage around which you place objects, lights, and cameras and is also where you will render final images or animations.
- Visual Model is where you create three-dimensional objects that you can then transfer to your scene in Renderize Live.
- Visual Font makes possible three-dimensional text objects with beveling and extruding properties.
- Visual Image is a two-dimensional image editor.
- Visual Catalog organizes projects from the preceding modules and facilitates the sharing of resources within the suite.

Visual Catalog is a new feature with Visual Reality 1.5 and it's a welcome addition that makes it much easier to carry on a project between the different applications. Also the handbooks for each module, previously available separately, have been combined into one integrated user's manual. It's well written and helpful, filled with pointers for optimizing performance as well as aesthetic tips. The tutorials are in depth, explaining each step of the process rather than just providing a quick run-through.

What Visual Software invites is the creation of virtual worlds, and toward this end Visual Reality's set design tools are impressive. The Light Designer in Renderize Live provides complete control over light and shadow: define ambient lighting, point lights, spot lights, and area lights; specify color, intensity, and attenuation, as well as position and direction; shadows can be enabled or disabled for each light, and the intensity and sharpness of the shadow set by the user. Photorealistic effects are enhanced by adjustable distance haze/fog levels and colors, and by camera settings that simulate the look of zoom and wide-angle lenses. Cameras are also fully animatable (though it is important to note that objects are not animatable). Materials editing and mapping capabilities are extensive, not to mention that the package comes with over 1,000 textures, including 200 seamless textures. Rendering options allow for texture, bump, reflection, and transparency mapping.

Visual Model, the arena for three-dimensional object creation, is also packed with features including Boolean

Create your own virtual worlds with Visual Reality or sample one of theirs. The Northern Castle is just one royalty-free three-dimensional environment available to game developers in the Simply Scenes series from Visual Software.

operations for adding, subtracting, and intersecting forms, along with the usual editing tools. Unfortunately, I haven't found it very easy to make use of all these goodies. Its method of defining three-dimensional space is not very intuitive, so that I'm always having to refer to the manual to make sense of the process. And the worlds I've created with Visual Reality are lonely places, given how difficult I've found it to attempt organic forms in the modeler.

Speaking of lonely virtual worlds, Visual Software is introducing a companion series called Simply Scenes that consists of fully rendered yet eerily uninhabited royalty-free three-dimensional settings for game developers to use. You can take elements of the three-dimensional environment and modify, move, retexture, or even remove them for use elsewhere. The idea is for designers to place their own characters and objects within these ready-made scenes. A hauntingly beautiful fortress, The Northern Castle, is one of the first. There's also a soup-to-nuts assemblage of everyday settings such as a dining room, a beach, and a country club locker room. On deck are Western Town, the Solar System, and Lunar and Sea Colonies. You can wander alone through each of these exotic locales for a mere $49 per.

3D For You Too

We've looked at three significantly different programs, each with its own distinct advantages and disadvantages. And the marketplace is crowded with many more than I could profile here even if I'd had the chance to use them all, which I haven't...yet.

If you've been thinking about making the move to three-dimensional design, think some more about just what it is you hope to find in those three dimensions. As these three examples show, there's a wide range of prices, features, and functionality to consider. Asked to choose between them, I'm not sure I could. I'd hate to give up the depth afforded by 3D Studio, I love the natural feel of working in trueSpace, and if I was just starting with three-dimensional graphics the bang-for-the-buck value of Visual Reality would surely be tempting. Fortunately, certain file formats are common between programs, so if you do have access to a variety of three-dimensional software you can probably make use of the best features of each.

If you've been using one three-dimensional program and you're not satisfied with what you've been able to do with it, there may be a better choice out there for you. It doesn't necessarily mean spending more money this time around, either. Rather, it's that quest for the brush that fits your hand. ■

David Sieks is a contributing editor to Game Developer. *Contact him via e-mail at dsieks@arnarb.harvard.edu or through* Game Developer *magazine.*